Machine Learning Engineering for Production (MLOps) Specialization

by DeepLearning.AI

Course #2 Machine Learning Data Lifecycle in Production

Course Site

Made By: Matias Borghi

Table of Contents

Summary

Week 1: Collecting, Labeling and Validating Data

This week covers a quick introduction to machine learning production systems. More concretely you will learn about leveraging the TensorFlow Extended (TFX) library to collect, label and validate data to make it production ready.

Learning Objectives

- **Describe the differences between ML modeling and a production ML system**
- **Identify responsible data collection for building a fair production ML system**
- **Discuss data and concept change and how to address it by annotating new training data with direct labeling and/or human labeling**
- **Address training data issues by generating dataset statistics and creating, comparing and updating data schemas**

Week 2: Feature Engineering, Transformation and Selection

Implement feature engineering, transformation, and selection with TensorFlow Extended by encoding structured and unstructured data types and addressing class imbalances

Learning Objectives

- **Define a set of feature engineering techniques, such as scaling and binning**
- **Use TensorFlow Transform for a simple preprocessing and data transformation task**
- **Describe feature space coverage and implement different feature selection methods**
- **Perform feature selection using scikit-learn routines and ensure feature space coverage**

Week 3: Data Journey and Data Storage

Understand the data journey over a production system's lifecycle and leverage ML metadata and enterprise schemas to address quickly evolving data.

Learning Objectives

- **Describe data journey through data lineage and provenance**
- **Integrate the sequence of pipeline artifacts into metadata storage using ML Metadata library**
- **Iteratively create enterprise data schema**

- **Explain how to integrate enterprise data into feature stores, data warehouses and data lakes**

Week 4 (Optional): Advanced Labeling, Augmentation and Data Preprocessing

Combine labeled and unlabeled data to improve ML model accuracy and augment data to diversify your training set.

Learning Objectives

- **Discuss direct, semi-supervised, weak supervision and active learning methods for labeling data**
- **Increase the diversity of your training set by data augmentation**
- **Perform advanced data preparation and transformation on different structured and unstructured data types**

Week 1: Collecting, Labeling and Validating Data

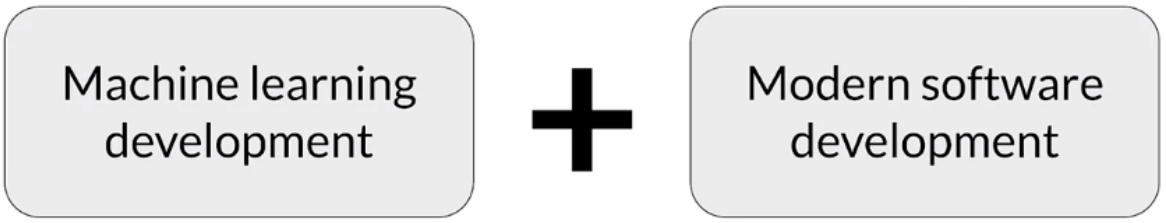
Introduction to Machine Learning Engineering in Production

Overview

ML modeling vs production ML

	Academic/Research ML	Production ML
Data	Static	Dynamic - Shifting
Priority for design	Highest overall accuracy	Fast inference, good interpretability
Model training	Optimal tuning and training	Continuously assess and retrain
Fairness	Very important	Crucial
Challenge	High accuracy algorithm	Entire system

Production machine learning



Managing the entire life cycle of data

- Labeling
- Feature space coverage
- Minimal dimensionality
- Maximum predictive data
- Fairness
- Rare conditions

Modern software development

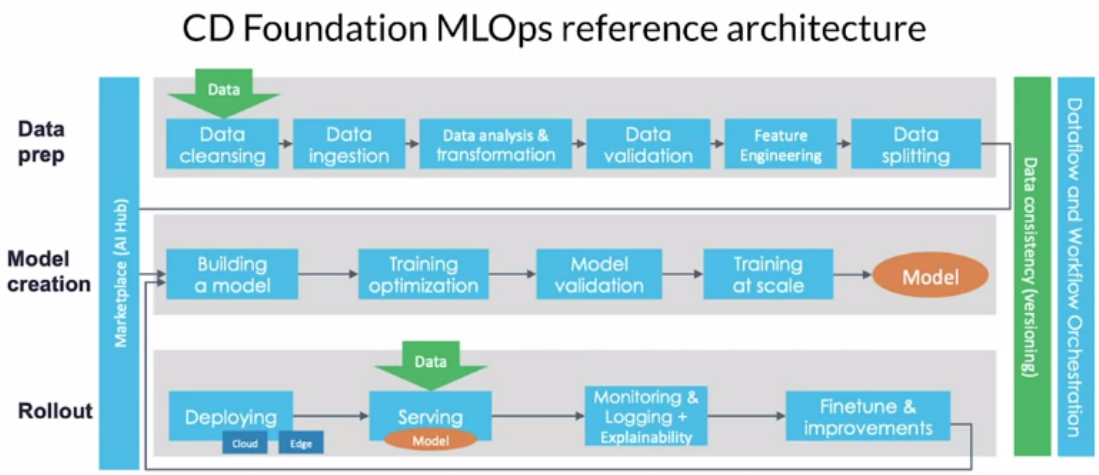
Accounts for:

- Scalability
- Extensibility
- Configuration
- Consistency & reproducibility
- Safety & security
- Modularity
- Testability
- Monitoring
- Best practices

Challenges in production grade ML

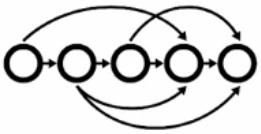
- Build integrated ML systems
- Continuously operate it in production
- Handle continuously changing data
- Optimize compute resource costs

Production ML infrastructure

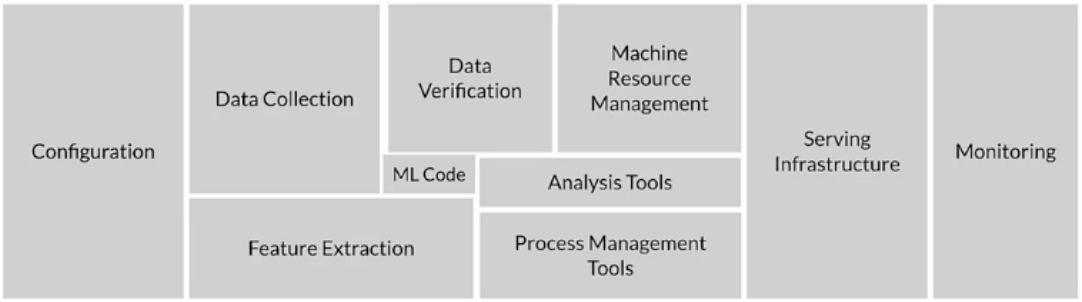


Directed acyclic graphs

- A directed acyclic graph (DAG) is a directed graph that has no cycles
- ML pipeline workflows are usually DAGs
- DAGs define the sequencing of the tasks to be performed, based on their relationships and dependencies.

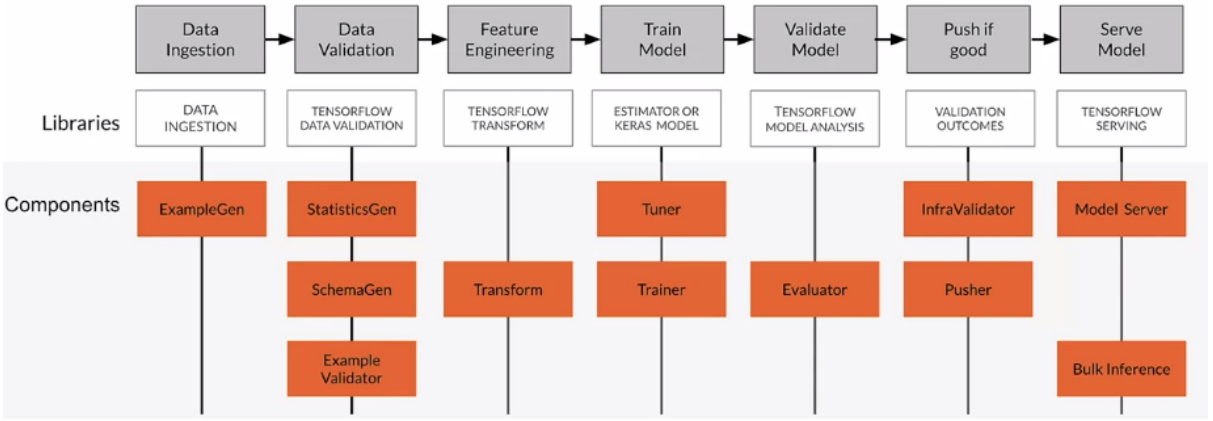


Pipeline orchestration frameworks

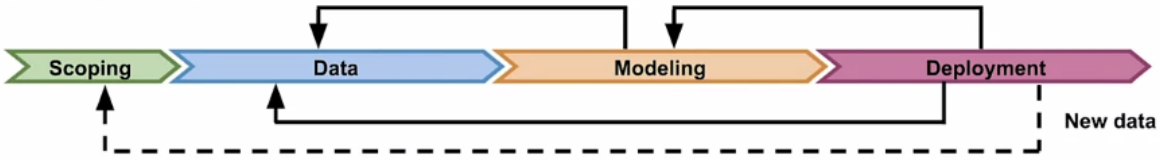


- Responsible for scheduling the various components in an ML pipeline DAG dependencies
- Help with pipeline automation
- Examples: Airflow, Argo, Celery, Luigi, Kubeflow

TFX production components



Key points



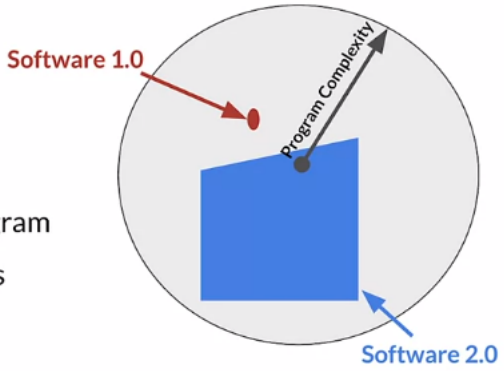
- Production ML pipelines: automating, monitoring, and maintaining end-to-end processes
- Production ML is much more than just ML code
  - ML development + software development
- TFX is an open-source end-to-end ML platform

Collecting Data

Importance of Data

ML: Data is a first class citizen

- Software 1.0
  - Explicit instructions to the computer
- Software 2.0
  - Specify some goal on the behavior of a program
  - Find solution using optimization techniques
  - Good data is key for success
  - Code in Software = Data in ML



Everything starts with data

- Models aren't magic
- Meaningful data:
  - maximize predictive content
  - remove non-informative data
  - feature space coverage

Garbage in, garbage out

$$f(\text{🗑}) = \text{🗑}$$

Key Points

- Understand users, translate user needs into data problems
- Ensure data coverage and high predictive signal
- Source, store and monitor quality data responsibly

Example Application: Suggesting Runs

Example application: Suggesting runs

Users	Runners
User Need	Run more often
User Actions	Complete run using the app
ML System Output	• What routes to suggest • When to suggest them
ML System Learning	• Patterns of behaviour around accepting run prompts • Completing runs • Improving consistency

Key considerations

- Data availability and collection
  - What kind of/how much data is available?
  - How often does the new data come in?
  - Is it annotated?
    - If not, how hard/expensive is it to get it labeled?
- Translate user needs into data needs
  - Data needed
  - Features needed
  - Labels needed

Example dataset

		FEATURES				
	Runner ID	**Run**	**Runner Time**	**Elevation**	**Fun**	
EXAMPLES	AV3DE	Boston Marathon	03:40:32	1,300 ft	Low	LABELS
	X8KGF	Seattle Oktoberfest 5k	00:35:40	0 ft	High	
	BH9IU	Houston Half-marathon	02:01:18	200 ft	Medium	

Get to know your data

- Identify data sources
- Check if they are refreshed
- Consistency for values, units, & data types
- Monitor outliers and errors

Translate user needs into data needs

Data Needed	• Running data from the app • Demographic data • Local geographic data

Features Needed	• Runner demographics • Time of day • Run completion rate • Pace • Distance ran • Elevation gained • Heart rate

Labels Needed	• Runner acceptance or rejection of app suggestions • User generated feedback regarding why suggestion was rejected • User rating of enjoyment of recommended runs

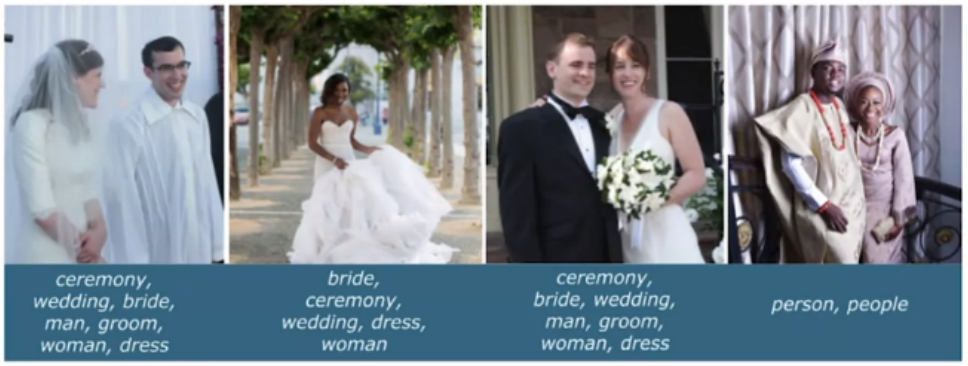
Key points

- Understand your user, translate their needs into data problems
  - What kind of/how much data is available
  - What are the details and issues of your data
  - What are your predictive features
  - What are the labels you are tracking
  - What are your metrics

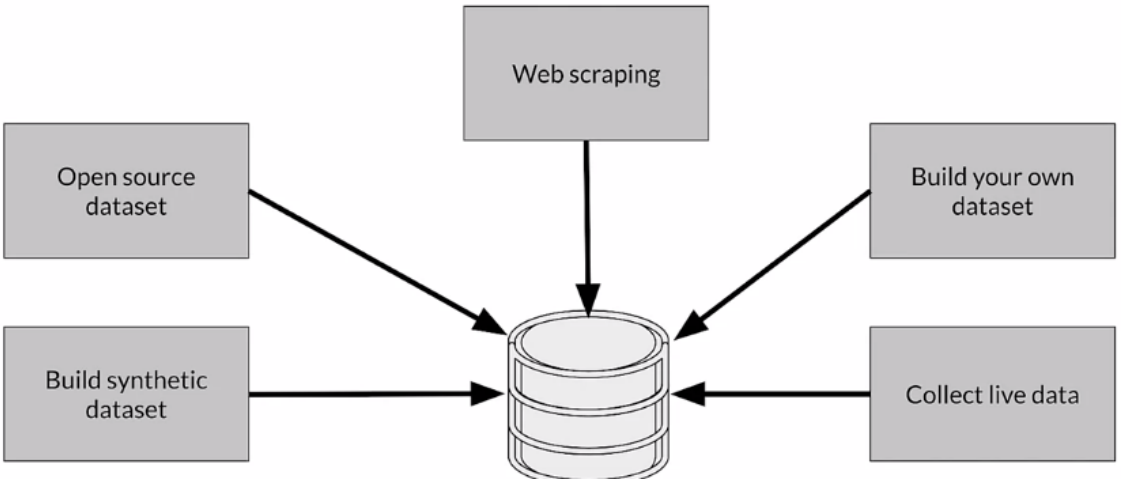
Responsible Data: Security, Privacy & Fairness

Avoiding problematic biases in datasets

Example: classifier trained on the Open Images dataset



Source Data Responsibly



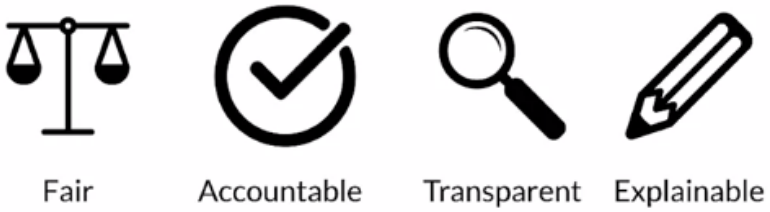
Data security and privacy

- Data collection and management isn't just about your model
  - Give user control of what data can be collected
  - Is there a risk of inadvertently revealing user data?
- Compliance with regulations and policies (e.g. GDPR)

Users privacy

- Protect personally identifiable information
  - Aggregation - replace unique values with summary value
  - Redaction - remove some data to create less complete picture

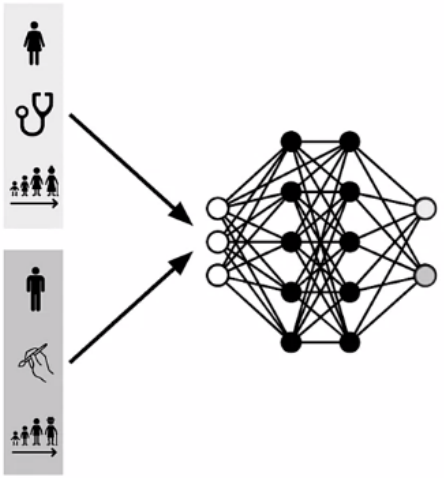
How ML systems can fail users



- Representational harm
- Opportunity denial
- Disproportionate product failure
- Harm by disadvantage

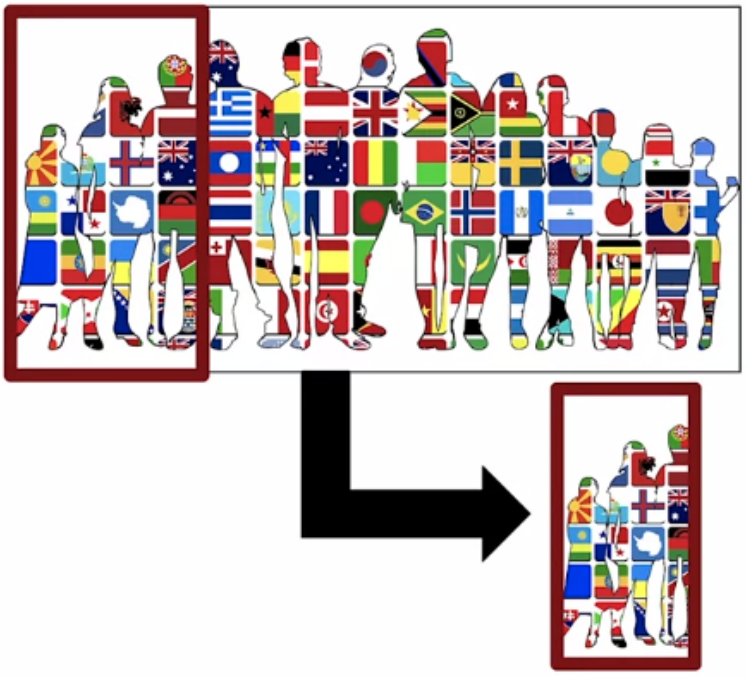
Some of the ways that ML Systems can fail are through things like representational harm. So **representational harm** is where a system will amplify or reflect a negative stereotype about particular groups. **Opportunity denial** is when a system makes predictions that have negative real life consequences that could result in lasting impacts. **Disproportionate product failure** is where the effectiveness of your model is really skewed so that the outputs happen more frequently for particular groups of users, you get skewed outputs more frequently essentially can think of as errors more frequently. **Harm by disadvantage** is where a system will infer disadvantageous associations between different demographic characteristics and the user behaviors around that.

Commit to fairness



- Make sure your models are fair
  - Group fairness, equal accuracy
- Bias in human labeled and/or collected data
- ML Models can amplify biases

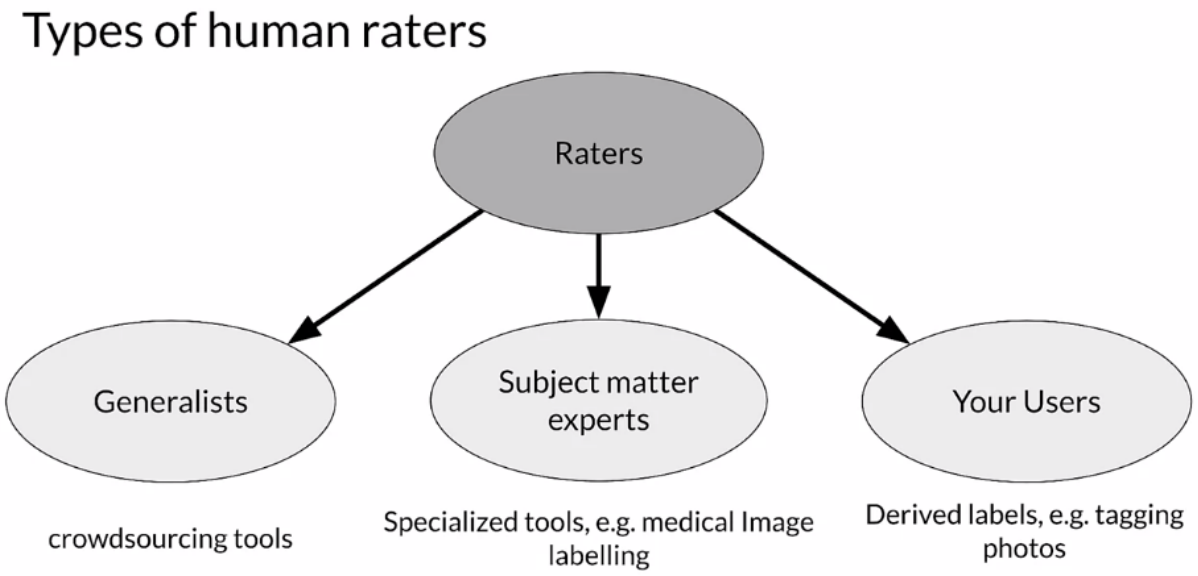
Biased data representation



Reducing bias: Design fair labeling systems

- Accurate labels are necessary for supervised learning

- Labeling can be done by:
  - Automation (logging or weak supervision)
  - Humans (aka “Raters”, often semi-supervised)



Key points

- Ensure rater pool diversity
- Investigate rater context and incentives
- Evaluate rater tools
- Manage cost
- Determine freshness requirements

Labeling Data

Case Study: Degraded Model Performance

To illustrate some of the issues with labeled data in performance settings, let's take a look at a case study.

So imagine that you're an online retailer and you're selling shoes and you have a model that predicts click through rates which helps you to decide how much inventory to order.

Why? You trained your model? You did fairly well in your metrics and so forth. Why is it different? What changed? Why does your model not predict men's dress shoes well now? And perhaps more importantly, how do you even know that you have a problem?

if you don't put good practices into place in the production setting, you're probably going to find out either when you order way too many shoes or not enough shoes. And that's not a situation that you want to be in in a business. This is going to cost you money.

Case study: taking action

- How to detect problems early on?
- What are the possible causes?
- What can be done to solve these?

What causes problems?

Kinds of problems:

- Slow - example: drift
- Fast - example: bad sensor, bad software update

Gradual problems

Data changes	• Trend and seasonality • Distribution of features changes • Relative importance of features changes
World changes	• Styles change • Scope and processes change • Competitors change • Business expands to other geos

So, maybe last year black shoes were really fashionable for men's dress shoes and now it's brown shoes or what have you.

Sudden problems

Data collection problem	Systems problem
• Bad sensor/camera • Bad log data • Moved or disabled sensors/cameras	• Bad software update • Loss of network connectivity • System down • Bad credentials

Why “Understand” the model?

- Mispredictions do not have uniform **cost** to your business
- The **data you have** is rarely the data you wish you had
- Model objective is nearly always a **proxy** for your business objectives
- Some percentage of your customers may have a **bad experience**

The real world does not stand still!

In the case of the shoes that we just looked at, we were predicting click through rates as a proxy for deciding how much inventory to order.

Data and Concept Change in Production ML

Detecting problems with deployed models

- Data and scope changes
- Monitor models and validate data to find problems early
- Changing ground truth: **label** new training data

Easy problems

- Ground truth changes slowly (months, years)
- Model retraining driven by:
  - Model improvements, better data
  - Changes in software and/or systems
- Labeling
  - Curated datasets
  - Crowd-based

Harder problems

- Ground truth changes faster (weeks)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Crowd-based

Really hard problems

- Ground truth changes very fast (days, hours, min)
- Model retraining driven by:
  - **Declining model performance**
  - Model improvements, better data
  - Changes in software and/or system
- Labeling
  - Direct feedback
  - Weak supervision

Key points

- Model performance decays over time
  - Data and Concept Drift
- Model retraining helps to improve performance
  - Data labeling for changing ground truth and scarce labels

Data labeling

Variety of Methods

- Process Feedback (Direct Labeling)
- Human Labeling
- ~~Semi-Supervised Labeling~~
- ~~Active Learning~~
- ~~Weak Supervision~~

→ Practice later as advanced labeling methods

Data labeling

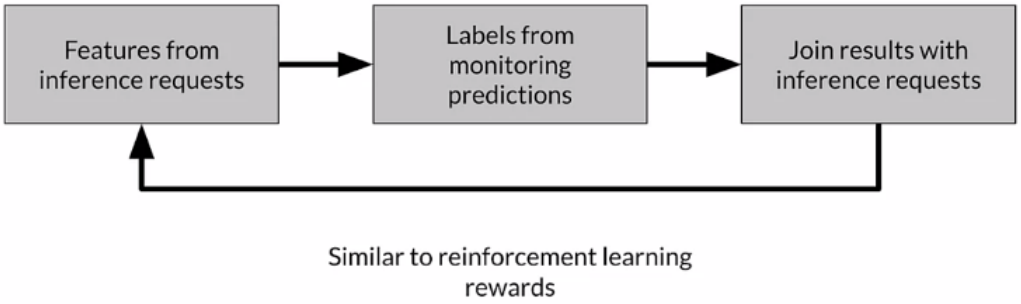
Process Feedback	Example: Actual vs predicted click-through
Human Labeling	Example: Cardiologists labeling MRI images

Actual versus predicted click-through rates. Suppose you have recommendations that you are giving to a user, did they actually click on the things that you recommend? If they did, you can label it positive, if they didn't you can label it negative. Human labeling, you can have humans look at data and apply labels to them. For example, you can ask cardiologists to look at MRI images and apply labels to them.

Why is labeling important in production ML?

- Using business/organisation available data
- Frequent model retraining
- Labeling ongoing and critical process
- Creating a training datasets requires labels

Direct labeling: continuous creation of training dataset



Labeling is an ongoing and often critical process in your application and your business. But at the end of the day, creating a training data set for supervised learning requires labels, you need to think about how you're going to do that. **Direct labeling**, which we'll talk about first or process feedback, is a way of continuously creating new training data that you're going to use to retrain your model. You're taking the features themselves from the inference requests that your model is getting. The predictions that your model is being asked to make and the features that are provided for that. You get labels for those inference requests by monitoring systems and using the feedback from those systems to label that data. One of the things that you need to solve there is to join the results that you get from monitoring those systems with the original inference request which could be hours or days apart. You might've run batches on Monday and you're getting feedback on Friday. You need to make sure that you can do those joins to apply those labels. In some ways you can think about this as similar to reinforcement learning, where instead of applying rewards based on action you're applying labels based on a prediction.

Process feedback - advantages

- Training dataset continuous creation
- Labels evolve quickly
- Captures strong label signals

Process feedback - disadvantages

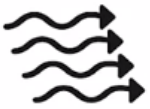
- Hindered by inherent nature of the problem
- Failure to capture ground truth
- Largely bespoke design

Process feedback - Open-Source log analysis tools

Logstash

Free and open source data processing pipeline

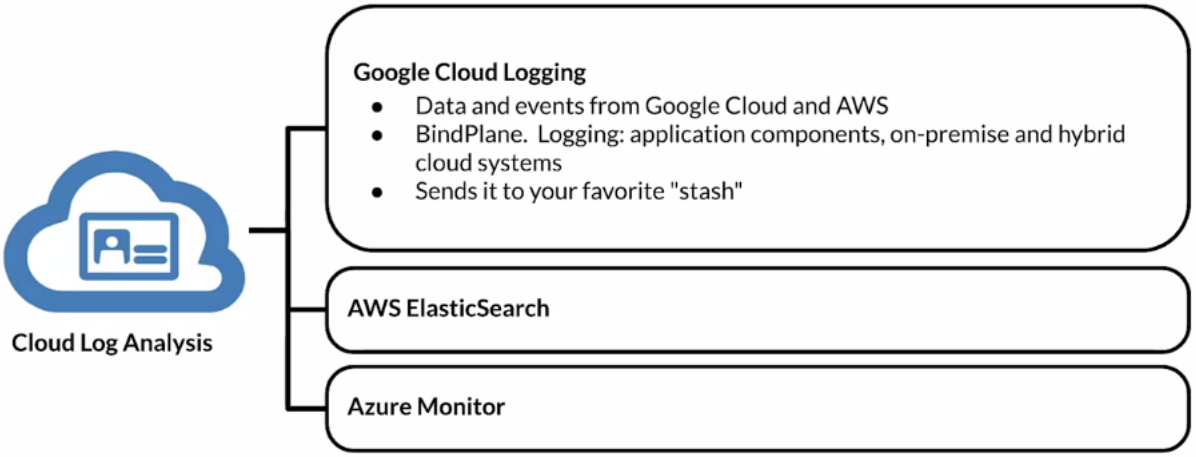
- Ingests data from a multitude of sources
- Transforms it
- Sends it to your favorite "stash."



Fluentd

Open source data collector
Unify the data collection and consumption

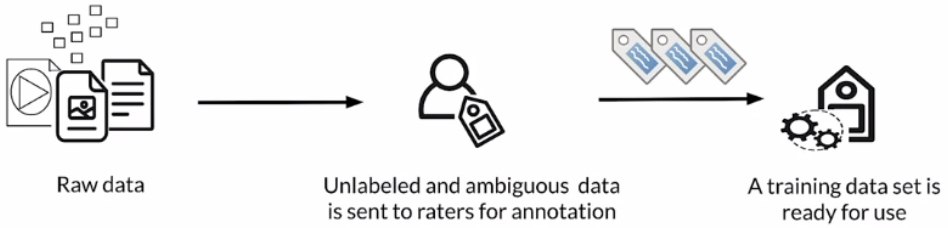
Process feedback - Cloud log analytics



Now, let's turn to human labeling. In human labeling, you have people, humans and we refer to those as raters. We ask them to examine data and assign labels to it. It sounds simple, it sounds like it might be painful but it's the way that a lot of data is generated and labeled. You start with raw data and you give it to people and you ask them to apply labels to it. That's the way that you create a training data set that you're going to use to train or retrain your model.

Human labeling

People (“raters”) to examine data and assign labels manually

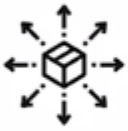


Human labeling - Methodology


Unlabeled data is collected


Human “raters” are recruited

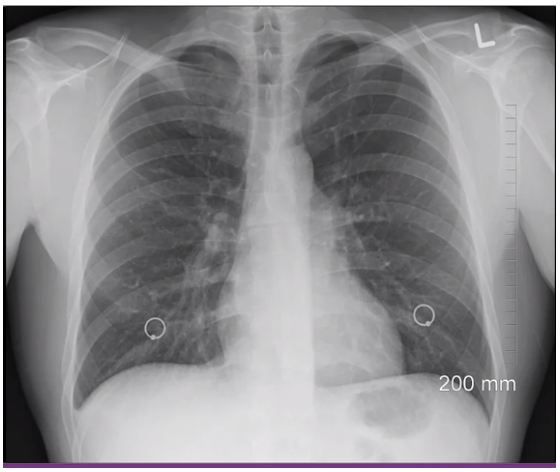

Instructions to guide raters are created


Data is divided and assigned to raters


Labels are collected and conflicts resolved

Human labeling - advantages

- More labels
- Pure supervised learning



One disadvantage, one issue here is that depending on the data that you have it can be very complex for a human to look at it and decide what the label should be. Something like this we might need a radiologist to look at this and tell us what the right labels should be which can be very expensive. But if you're also talking about situations where you have high dimensional data, it's very difficult for humans to look at, say, 100 different features and decide what the label is.

Human labeling - Disadvantages

- Quality consistency: Many datasets difficult for human labeling
- Slow
- Expensive
- Small dataset curation

Why is human labeling a problem?

Key points

- Various methods of data labeling
  - Process feedback
  - Human labeling

- Advantages and disadvantages of both

Validating Data

Detecting Data Issues

Drift and skew

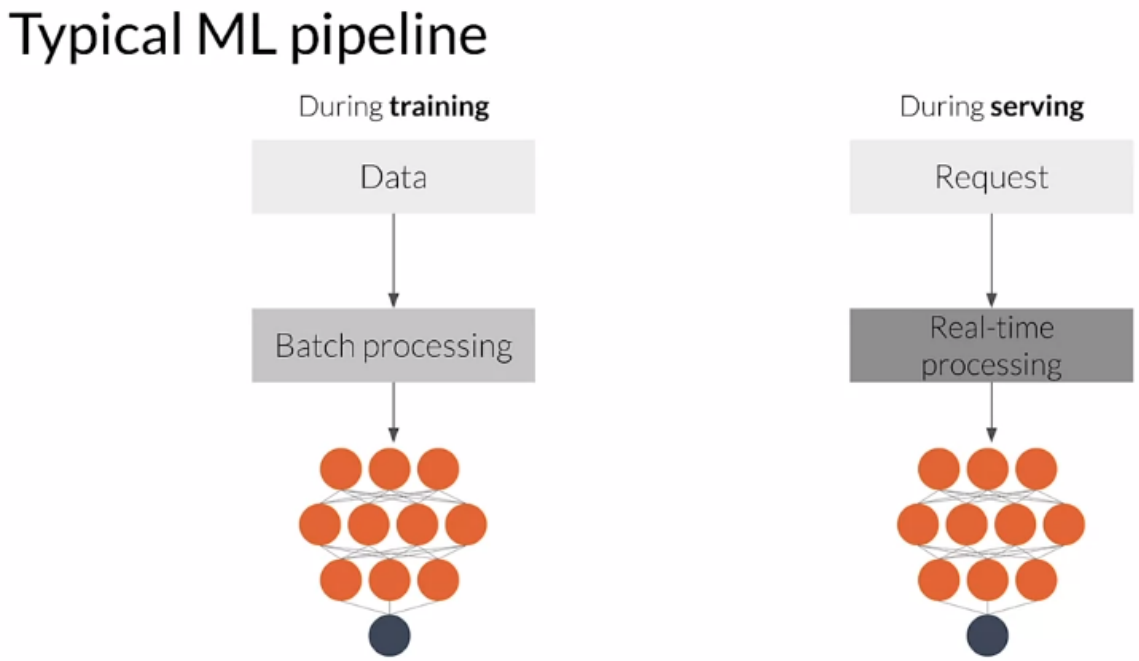
Drift

Changes in data over time, such as data collected once a day

Skew

Difference between two static versions, or different sources, such as training set and serving set

For example, it could be the difference between your training set and the data that you're getting for prediction requests, your serving set. Those differences are referred to as skew. In a typical ML pipeline, this shows batch processing, but it could also be online processing. You'll have different sources of data that are conceptually the same. They have the same feature vector, but over time they will change. That means that model performance can either drop quickly due to things like system failure or can decay over time due to changes in the data and things like changes in the world. We're going to focus on performance decay over time that arises due to issues between training and serving data. There's really two main reasons for that. There's **Data drift**, which are changes in the data between training and serving typically and Concept drift, which are changes in the world changes in the ground truth.



To understand model decay over time, an ML model will start to perform poorly in many cases and we refer to this as model decay. That's usually or often caused by drift, which is, changes in a conceptual way. There is, changes in the statistical properties of the features. It's sometimes due to things like seasonality or trend or unexpected events, or just changes in the world. For example, here, we're looking at an app that during training the app classified as a spammer, any user who is sending 20 or more messages per minute. We classified anybody like that as a spammer. But after a system update which you see as labeled on the chart there, both spammers and non-spammers start to send more messages. In this case, the data, the world has changed and that causes unwanted misclassification. We have all of our users classified as spammers which they probably won't like.

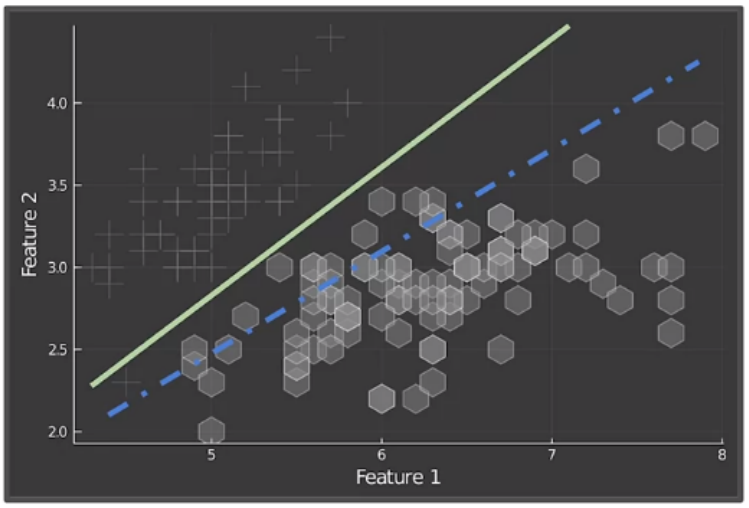
Model Decay : Data drift

Concept drift is a change in the statistical properties of the labels over time. At training, an ML model learns a mapping between the features and the labels. In a static world that's fine, that won't change. But in the real-world, distribution and the labels meaning will change. The model needs to change as well as the mapping found during training will no longer be valid.

Detecting data issues

- Detecting schema skew
  - Training and serving data do not conform to the same schema
- Detecting distribution skew
  - Dataset shift → covariate or concept shift
- Requires continuous evaluation

Performance decay : Concept drift



Let's take a look at a more rigorous definition of the drift and skew that we're talking about. **Dataset shift** occurs when the joint probability of x are features and y are labels is not the same during training and serving. The data has shifted over time. **Covariate shift** refers to the change in distribution of the input variables present in training and serving data. In other words, it's where the marginal distribution of x are features is not the same during training and serving, but the conditional distribution remains unchanged. **Concept shift** refers to a change in the relationship between the input and output variables as opposed to the differences in the Data Distribution or input itself. In other words, it's when the conditional distribution of y are labels given x are features is not the same during training and serving, but the marginal distribution of x are features remains unchanged.

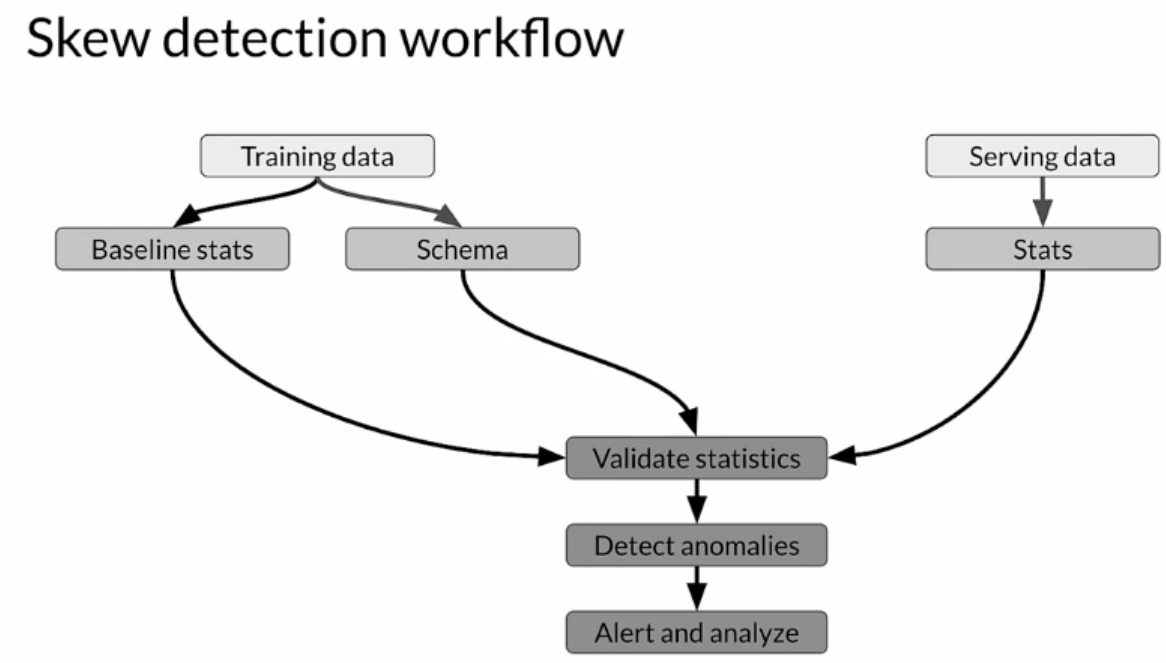
Detecting distribution skew

	Training	Serving
Joint	$P_{\text{train}}(y, x)$	$P_{\text{serve}}(y, x)$
Conditional	$P_{\text{train}}(y\|x)$	$P_{\text{serve}}(y\|x)$
Marginal	$P_{\text{train}}(x)$	$P_{\text{serve}}(x)$

Dataset shift $P_{\text{train}}(y, x) \neq P_{\text{serve}}(y, x)$

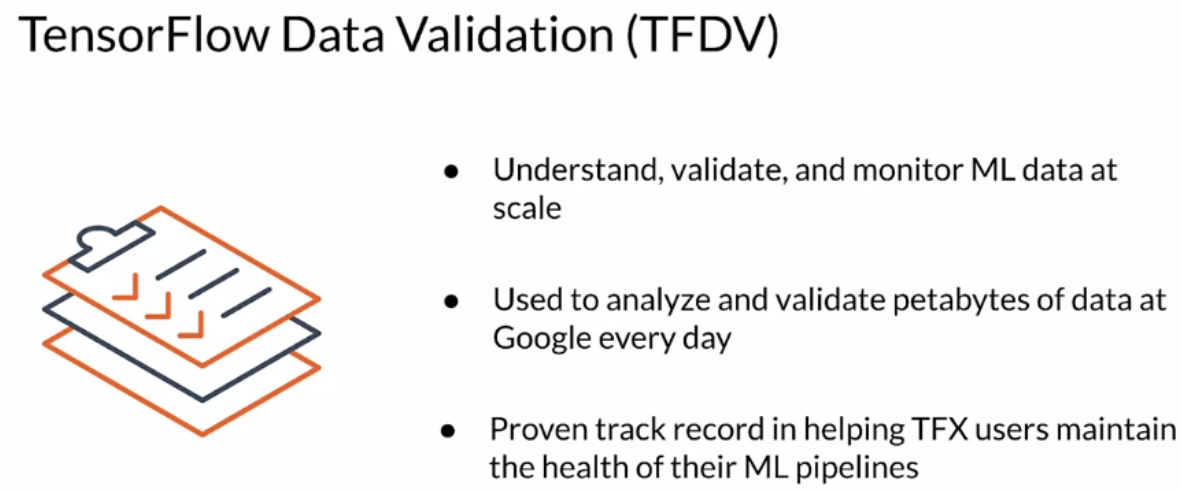
Covariate shift $P_{\text{train}}(y|x) = P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) \neq P_{\text{serve}}(x)$

Concept shift $P_{\text{train}}(y|x) \neq P_{\text{serve}}(y|x)$, $P_{\text{train}}(x) = P_{\text{serve}}(x)$



There's a straightforward workflow to detect data skew. The first stage is looking at training data and computing baseline statistics and a reference schema. Then you do basically the same with your serving data, you're going to generate the descriptive statistics. Then you compare the two. You compare your serving baseline statistics and instances. You check for differences between that and your training data. You look for skew and drift. Significant changes become anomalies and they'll trigger an alert. That alert goes to whoever's monitoring system, that can either be a human or another system to analyze the change and decide on the proper course of action. That's got to be the remediation of the way that you're going to fix and react to that problem.

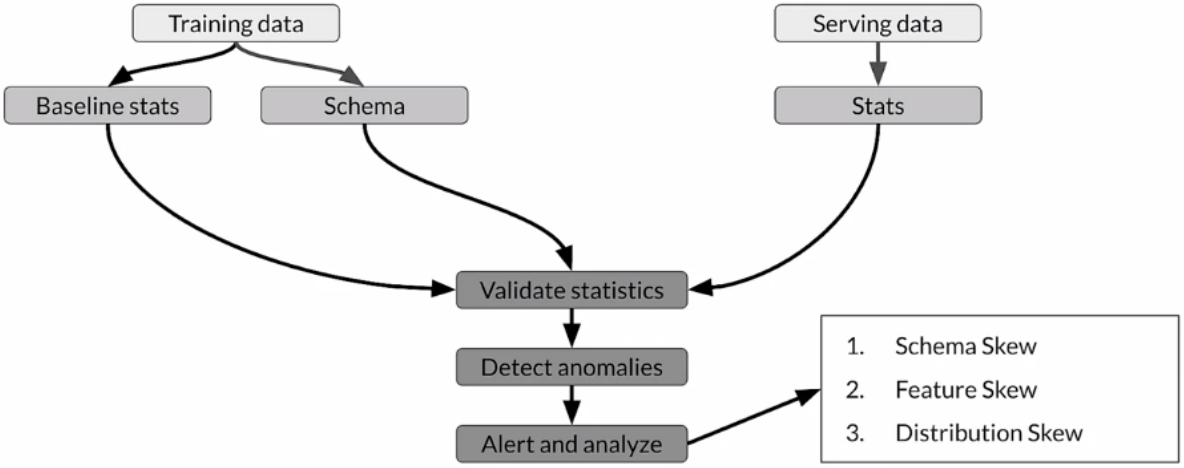
TensorFlow Data Validation



TFDV capabilities

- Generates data statistics and browser visualizations
- Infers the data schema
- Performs validity checks against schema
- Detects training/serving skew

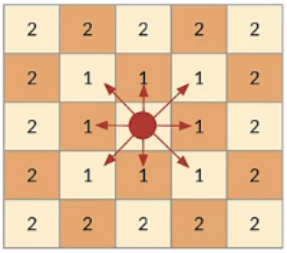
Skew detection - TFDV



Skew - TFDV

- Supported for categorical features
- Expressed in terms of L-infinity distance (Chebyshev Distance):

$$D_{\text{Chebyshev}}(x, y) = \max_i(|x_i - y_i|)$$



- Set a threshold to receive warnings

Schema skew

Serving and training data don't conform to same schema:

- For example, `int != float`

Feature skew

Training **feature values** are different than the serving **feature values:**

- Feature values are modified between training and serving time
- Transformation applied only in one of the two instances

Distribution skew

Distribution of serving and training dataset is significantly different:

- Faulty sampling method during training
- Different data sources for training and serving data
- Trend, seasonality, changes in data over time

Key points

- TFDV: Descriptive statistics at scale with the embedded facets visualizations
- It provides insight into:
  - What are the underlying statistics of your data
  - How does your training, evaluation, and serving dataset statistics compare
  - How can you detect and fix data anomalies

Wrap up

- Differences between ML modeling and a production ML system
- Responsible data collection for building a fair production ML system
- Process feedback and human labeling
- Detecting data issues

Practice data validation with TFDV in this week's exercise notebook

Test your skills with the programming assignment

Week 1 Optional References

Week 1: Collecting, Labeling and Validating Data

This is a compilation of optional resources including URLs and papers appearing in lecture videos. If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

MLops

Data 1st class citizen

Runners app

Rules of ML

Bias in datasets

Logstash

Fluentd

Google Cloud Logging

AWS ElasticSearch

Azure Monitor

TFDV

Chebyshev distance

Papers

Konstantinos, Katsiapis, Karmarkar, A., Altay, A., Zaks, A., Polyzotis, N., … Li, Z. (2020). Towards ML Engineering: A brief history of TensorFlow Extended (TFX). http://arxiv.org/abs/2010.02013

Paleyes, A., Urma, R.-G., & Lawrence, N. D. (2020). Challenges in deploying machine learning: A survey of case studies. http://arxiv.org/abs/2011.09926

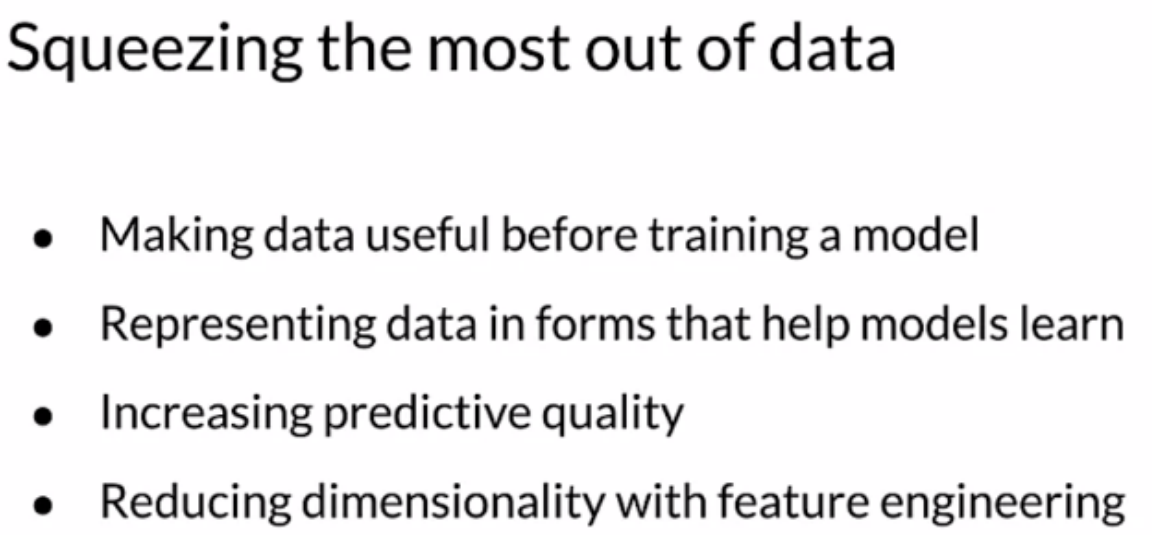
ML code fraction:

Sculley, D., Holt, G., Golovin, D., Davydov, E., & Phillips, T. (n.d.). Hidden technical debt in machine learning systems. Retrieved April 28, 2021, from Nips.cc https://papers.nips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf

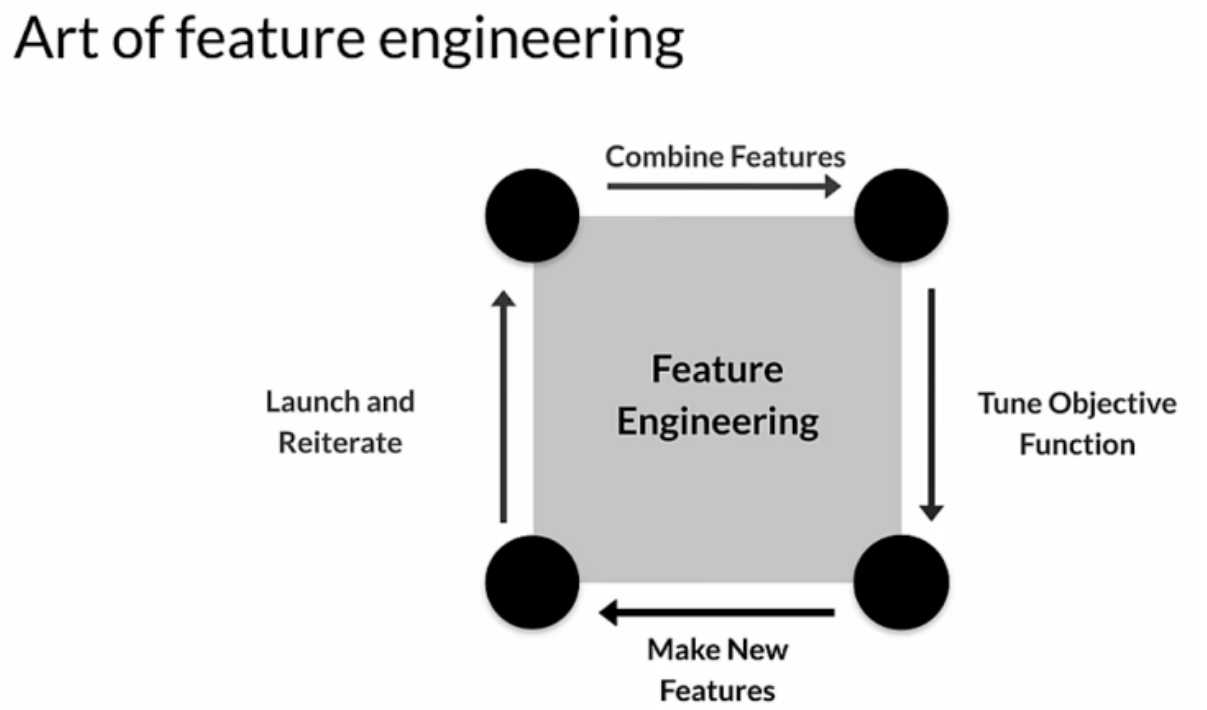
Week 2: Feature Engineering, Transformation and Selection

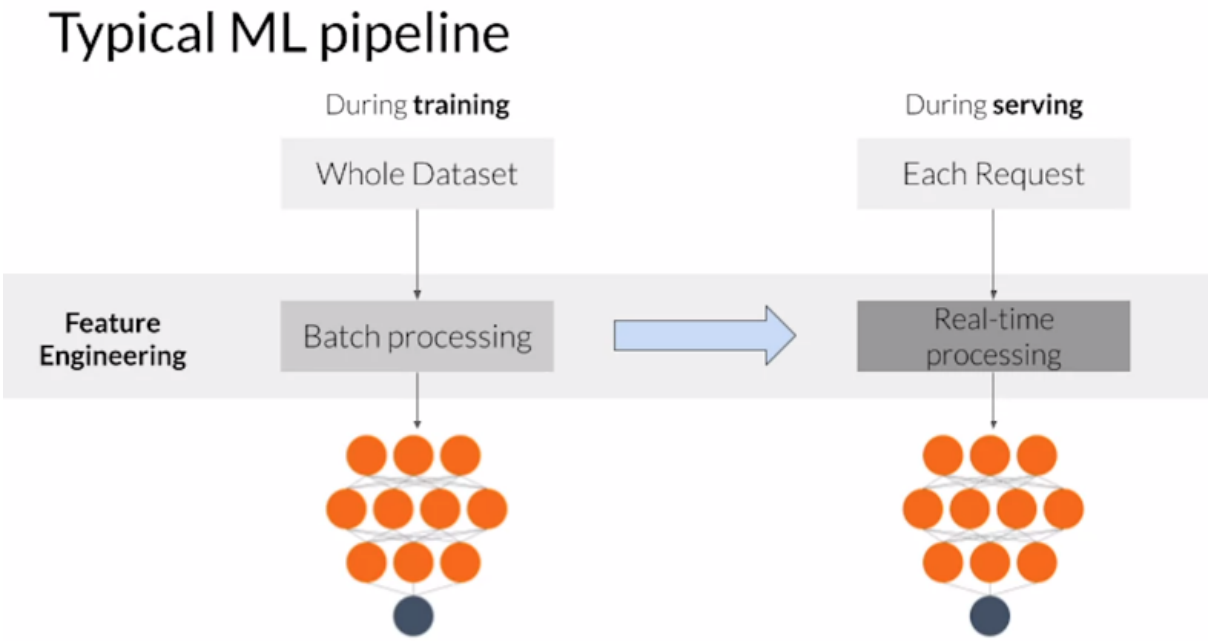
Feature Engineering

Introduction to Preprocessing



The art of feature engineering tries to improve your model's ability to learn while reducing, if possible, the compute resources it requires. It does this by transforming and projecting, eliminating and or combining the features in your raw data to form a new version of your data set. So typically across the ML pipeline, you incorporate the original features often transformed or projected to a new space and or combinations of your features.





Feature engineering is usually applied in two fairly different ways, during training, you usually have the entire data set available to you. So you can use global properties of individual features in your feature engineering transformations. For example, you can compute the standard deviation of a feature and then use that to perform normalization. However, when you serve your trained model, you must do the same feature engineering so that you give your model the same types of data that it was trained on. For example, if you created a one hot vector for a categorical feature when you trained, you need to also create an equivalent one hot vector when you serve your model. However, during training and serving, you typically process each request individually, so it's important that you include global properties of your features, such as the standard deviation. If you use it during training, include that with the feature engineering that you do when

serving, failing to do that right is a very common source of problems in production systems, and often these errors are difficult to find.

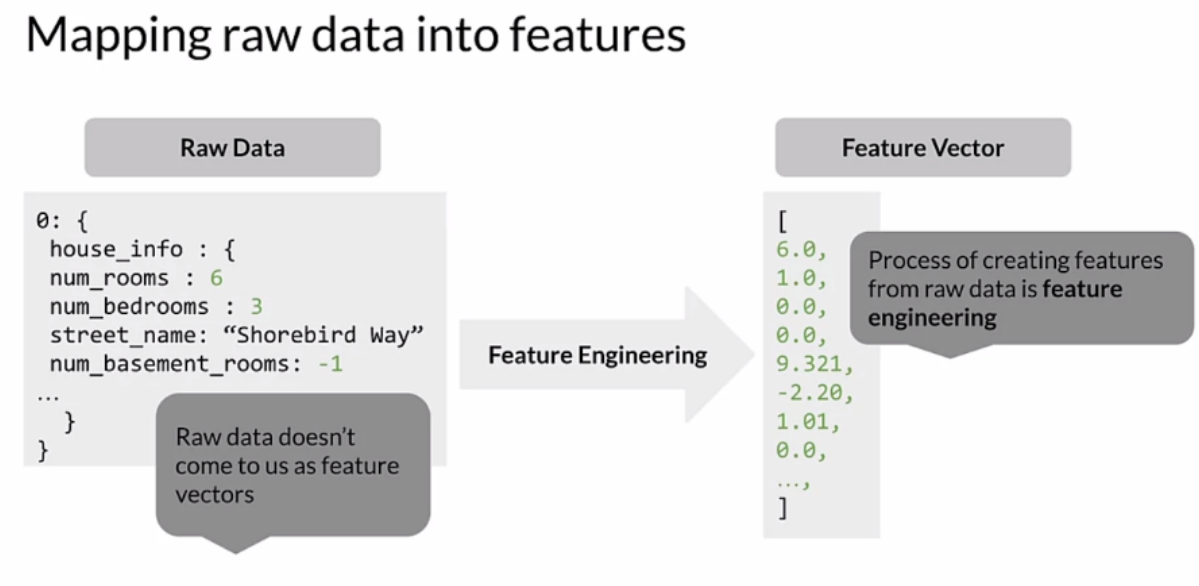
Key points

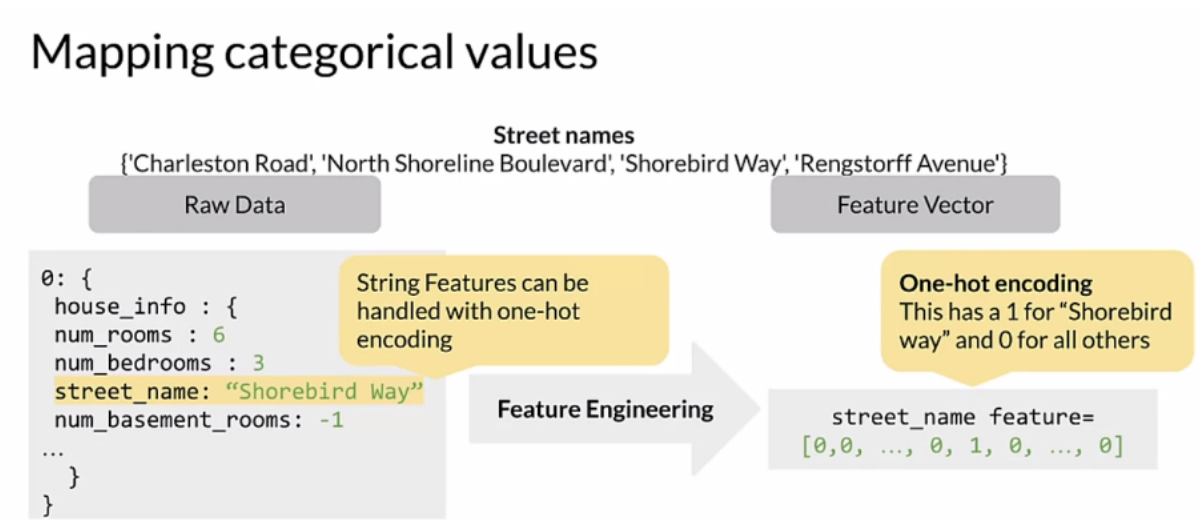
- Feature engineering can be difficult and time consuming, but also very important to success
- Squeezing the most out of data through feature engineering enables models to learn better
- Concentrating predictive information in fewer features enables more efficient use of compute resources
- Feature engineering during training must also be applied correctly during serving

Preprocessing Operations

Main preprocessing operations

One of the most important Preprocessing Operations is **Data cleansing**, which in broad terms consists in eliminating or correcting erroneous data. You'll often need to perform **Transformations** on your data, so scaling or normalizing your numeric values, for example. Since models, especially neural networks, are sensitive to the amplitude or range of numerical features, data preprocessing helps Machine Learning build better predictive models. **Dimensionality Reduction** involves reducing the number of features by creating lower dimension and more robust data representations. **Feature Construction** can be used to create new features by using several different techniques, which we'll talk about some of them.





Tensorflow provides three different functions for creating columns of categorical vocabulary and other frameworks do very similar things. A categorical column with vocabulary lists maps each string to an integer based on an explicit vocabulary list. Feature name is a string which corresponds to the categorical feature and vocabulary list is an ordered list that defines vocabulary. Feature or categorical column with a vocabulary file very similar from a file is used when you have two long lists and this function allows you to put the words in a separate file. In this case, vocabulary list is defined as a file that will define the list of words. This is very typical for working with vocabularies.

Categorical Vocabulary

```
# From a vocabulary list

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_list(

                            key=feature_name,

                            vocabulary_list=["kitchenware", "electronics", "sports"])
# From a vocabulary file

vocabulary_feature_column = tf.feature_column.categorical_column_with_vocabulary_file(

                             key=feature_name,

                             vocabulary_file="product_class.txt",

                             vocabulary_size=3)
```

Empirical knowledge of data

Text - stemming, lemmatization, TF-IDF, n-grams, embedding lookup

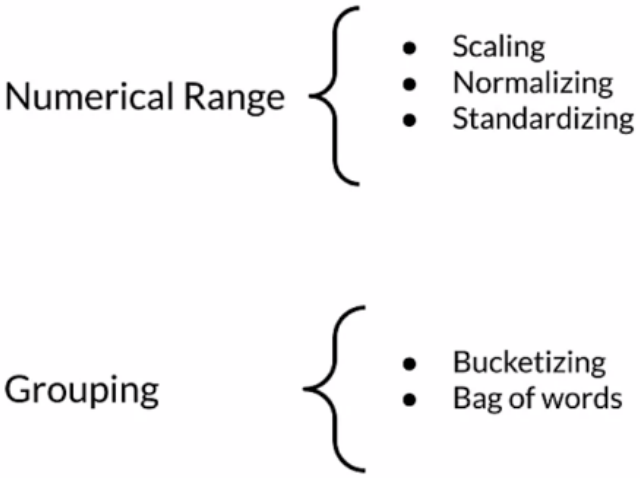
Images - clipping, resizing, cropping, blur, Canny filters, Sobel filters, photometric distortions

Key points

- Data preprocessing: transforms raw data into a clean and training-ready dataset
- Feature engineering maps:
  - Raw data into feature vectors
  - Integer values to floating-point values
  - Normalizes numerical values
  - Strings and categorical values to vectors of numeric values
  - Data from one space into a different space

Feature Engineering Techniques

Feature engineering techniques



Scaling

- Converts values from their natural range into a prescribed range
  - E.g. Grayscale image pixel intensity scale is [0,255] usually rescaled to [-1,1]

```
image = (image - 127.5) / 127.5
```

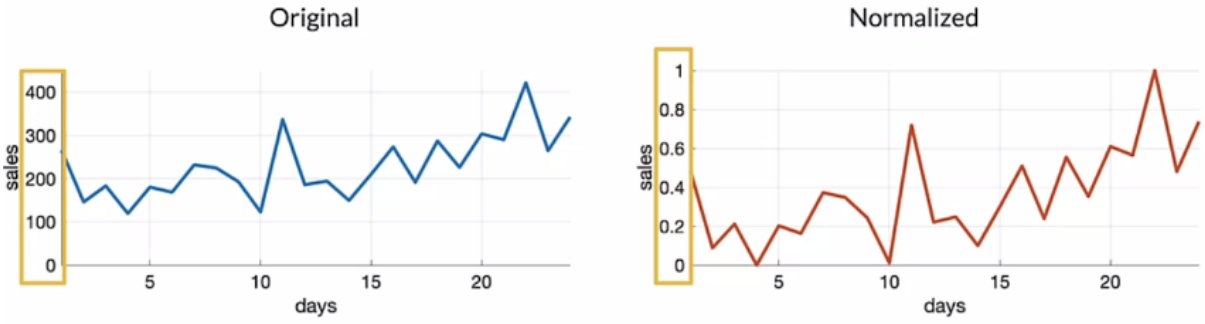
- Benefits
  - Helps neural nets converge faster
  - Do away with NaN errors during training
  - For each feature, the model learns the right weights

Normalizations are usually good if you know that the distribution of your data is not Gaussian. Doesn't always have to be true, but typically that's a good assumption to start with. A good rule of thumb. If you're working with data that you know is not Gaussian, or a normal distribution, then normalization is a good technique to start with. Normalization is widely used for scaling, you have a numerical feature that might start out like this, where numbers are falling and there is a kind of a range. You're going to transform that to the normalized version of that, which is bounded in a range between 0-1.

Normalization

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

$$X_{\text{norm}} \in [0, 1]$$

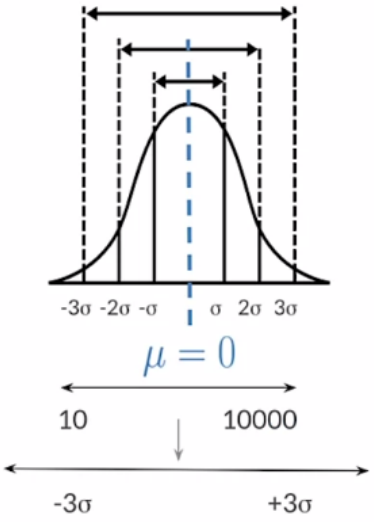


Standardization (z-score)

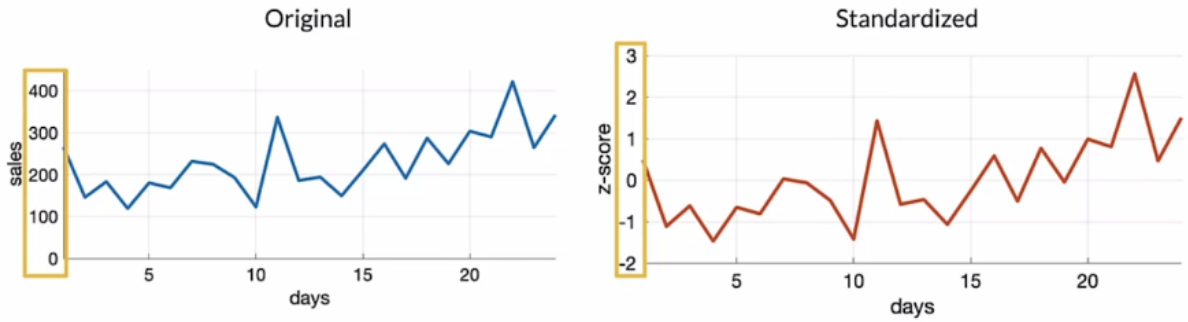
- Z-score relates the number of standard deviations away from the mean
- Example:

$$X_{\text{std}} = \frac{X - \mu}{\sigma} \quad \text{(z-score)}$$

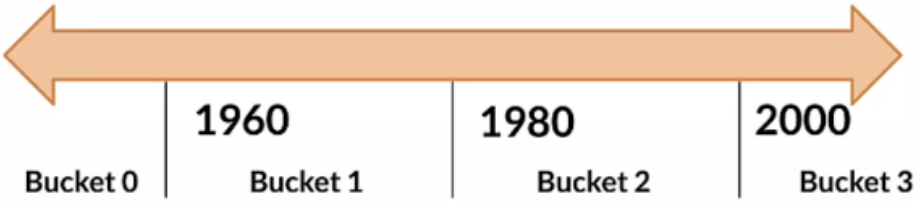
$$X_{\text{std}} \sim \mathcal{N}(0, \sigma)$$



Standardization, which is often using a Z-score, is a different technique. It's a way of scaling using the standard deviation. It's looking at the distribution itself and trying to scale relative to that. The example here is using, , and you're going to subtract the mean and divide by the standard deviation. That gives you the Z- score or the standardized value of X. Which is somewhere between zero and the standard deviation. This is how that's expressed, actually between some multiple of the standard deviation. But it's centered around the mean of the data. If the original looked like this, again, a standardized value of that might look like this. Notice that this score is centered on zero, so the mean is translated to zero. But you can have negative values and positive values that are beyond one. It's a little bit less bounded transformation than a normalization is. But there are some advantages to it, that your data is a normal distribution, then a standardization technique is a good place to start, it's a good rule of thumb to start with for your numerical features. But I encourage you to try both standardization and normalization and compare the results. Because sometimes it doesn't make much of a difference. Sometimes it can make a substantial difference. It's good to try both.

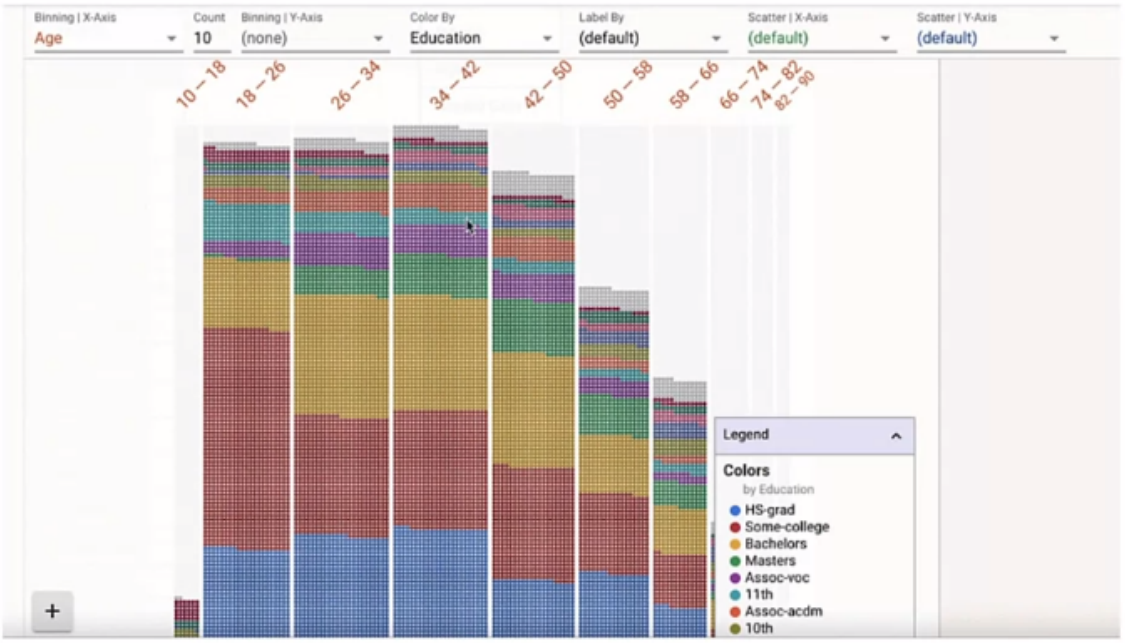


Bucketizing / Binning



Date Range	Represented as...
< 1960	[1, 0, 0, 0]
>= 1960 but < 1980	[0, 1, 0, 0]
>= 1980 but < 2000	[0, 0, 1, 0]
>= 2000	[0, 0, 0, 1]

Binning with Facets



Other techniques

Dimensionality reduction in embeddings

- Principal component analysis (PCA)
- t-Distributed stochastic neighbor embedding (t-SNE)
- Uniform manifold approximation and projection (UMAP)

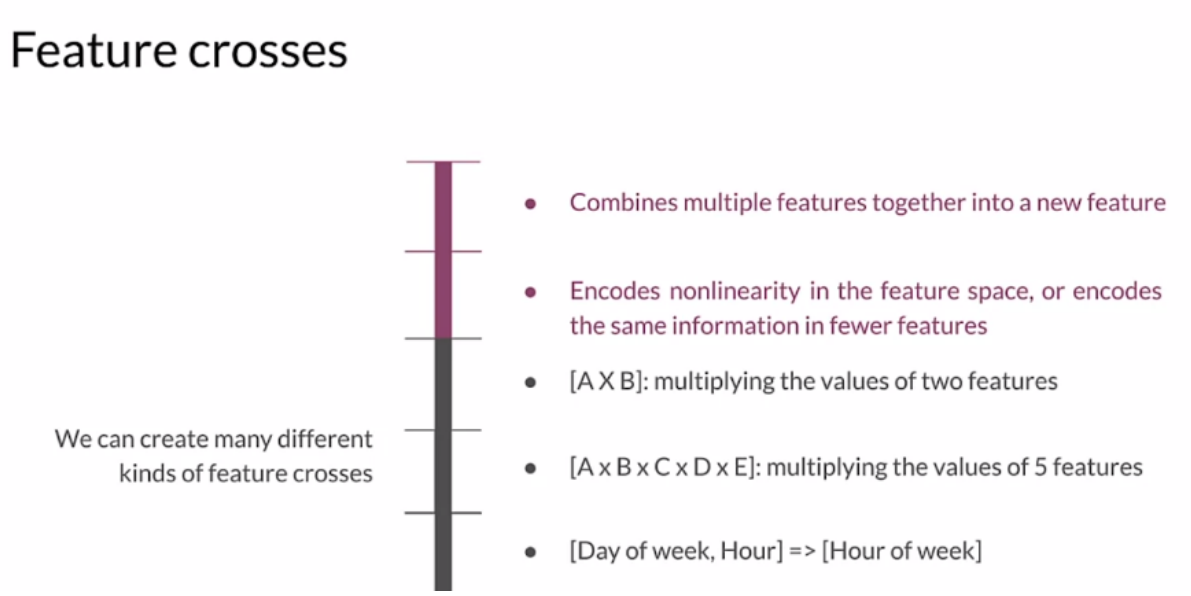
Feature crossing

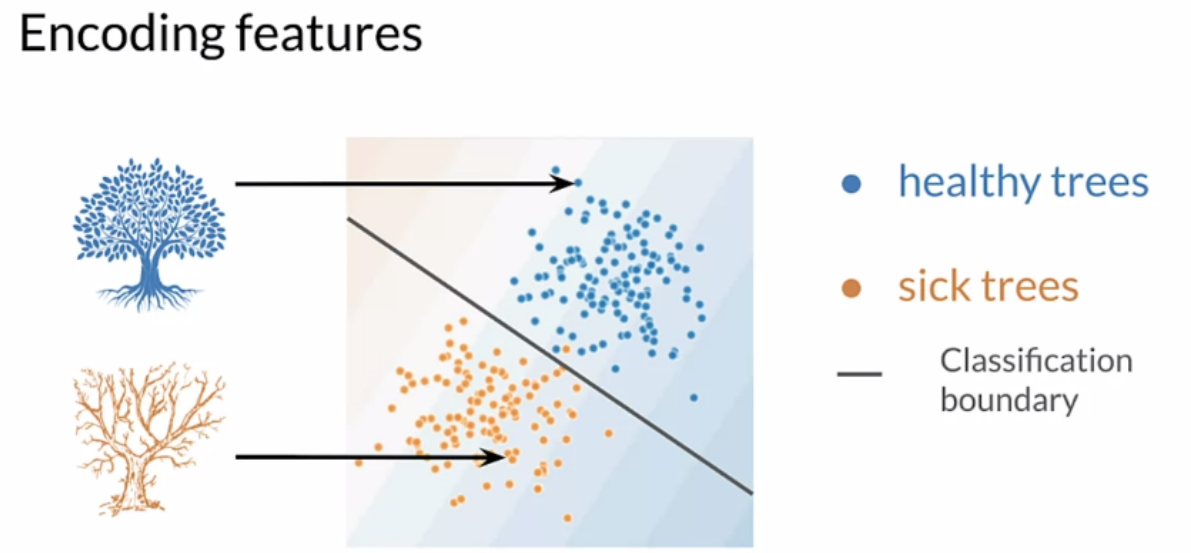
TensorFlow embedding projector

- Intuitive exploration of high-dimensional data
- Visualize & analyze
- Techniques
  - PCA
  - t-SNE
  - UMAP
  - Custom linear projections
- Ready to play

Key points

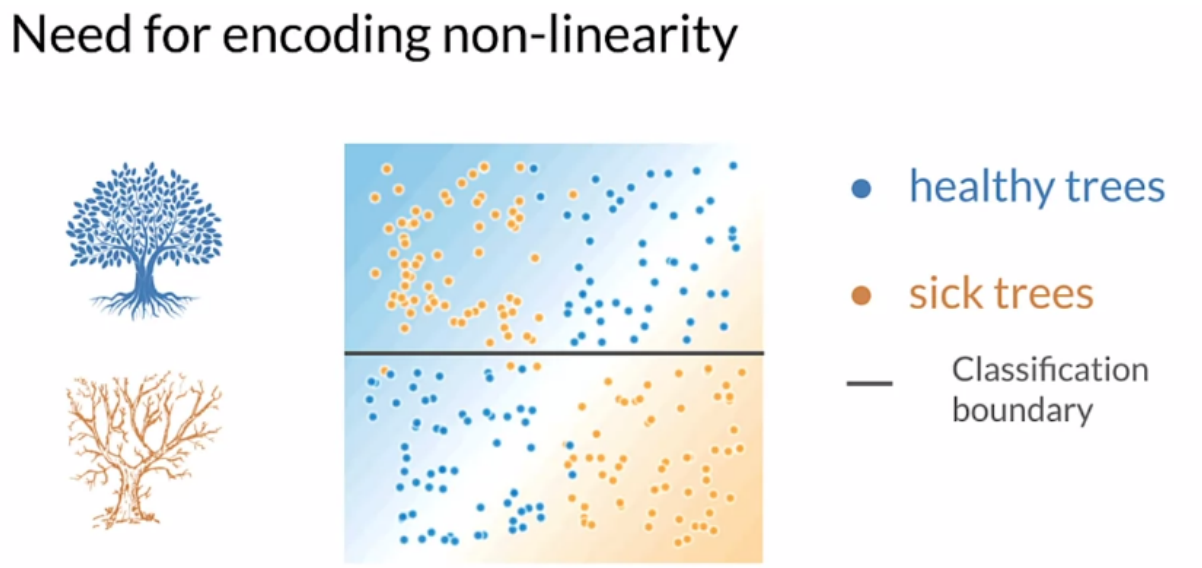
- Feature engineering:
  - Prepares, tunes, transforms, extracts and constructs features.
- Feature engineering is key for model refinement
- Feature engineering helps with ML analysis

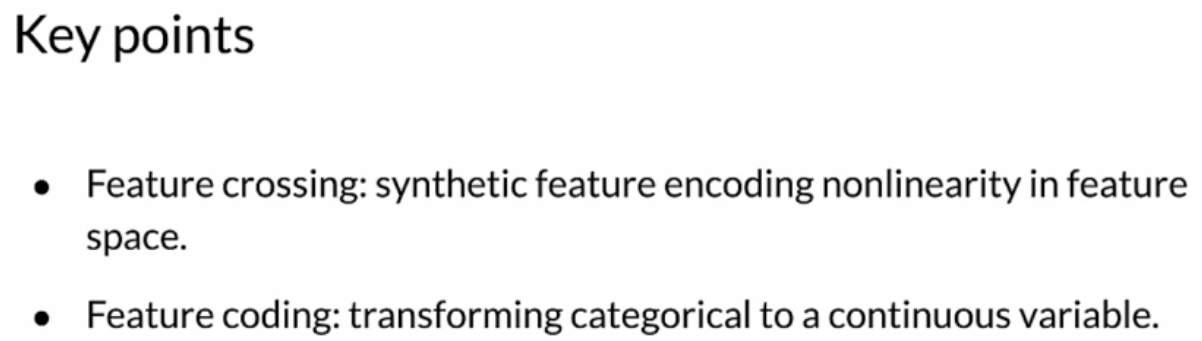




We're going to use a scatter-plot to try to understand our data. We're looking at the scatter plot and we use some visualization tools, and we ask ourselves, can we draw a line to separate these two groups, these two clusters? If we can use a line to create the decision boundary, then we know that we can use a linear model. That's great. In this case, looking at this, we could use a line to do that separation. That's great. Linear models are very efficient. We love that, but let's suppose the data looks like this. Not so easy anymore. This becomes a non-linear program. Then the question is, can we project this data into a linear space and use it with a linear model, or do we have to use a non-linear model to work with this data? Because if we try to draw just a linear classification boundary with the data as is, it doesn't really work too great. Again, we can use visualization tools to help us understand this. Looking at our data really helps inform us and guide the choices that we make as developers about what kind of models we choose and what feature engineering we apply. Key points here, Feature Crossing as a way to create synthetic

features, often encoding non-linearity in the features space. We're going to transform both categorical and numerical. We could do that in, into either continuous variables or the other way around.

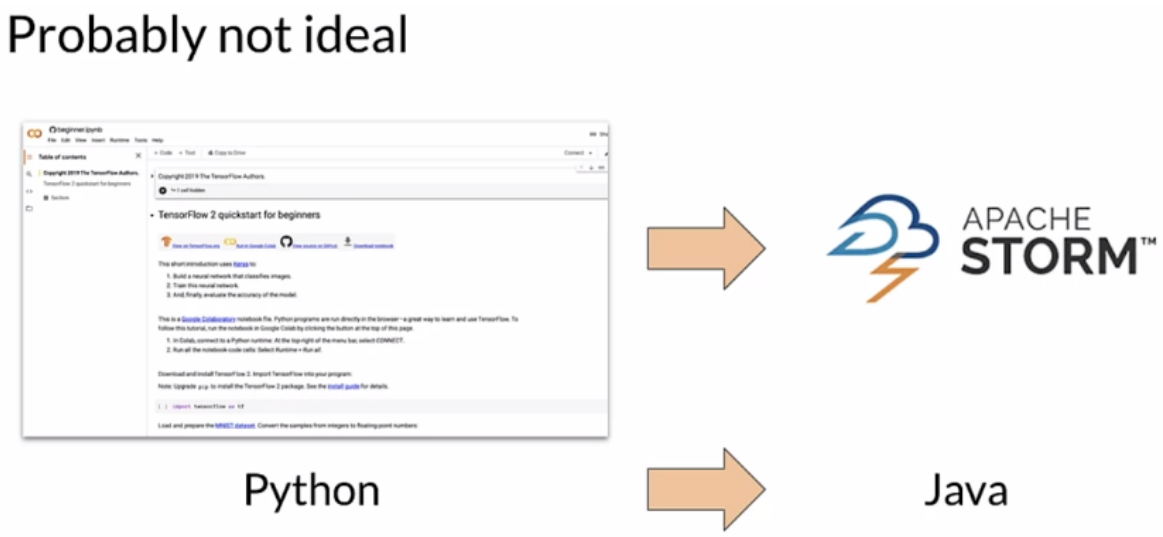




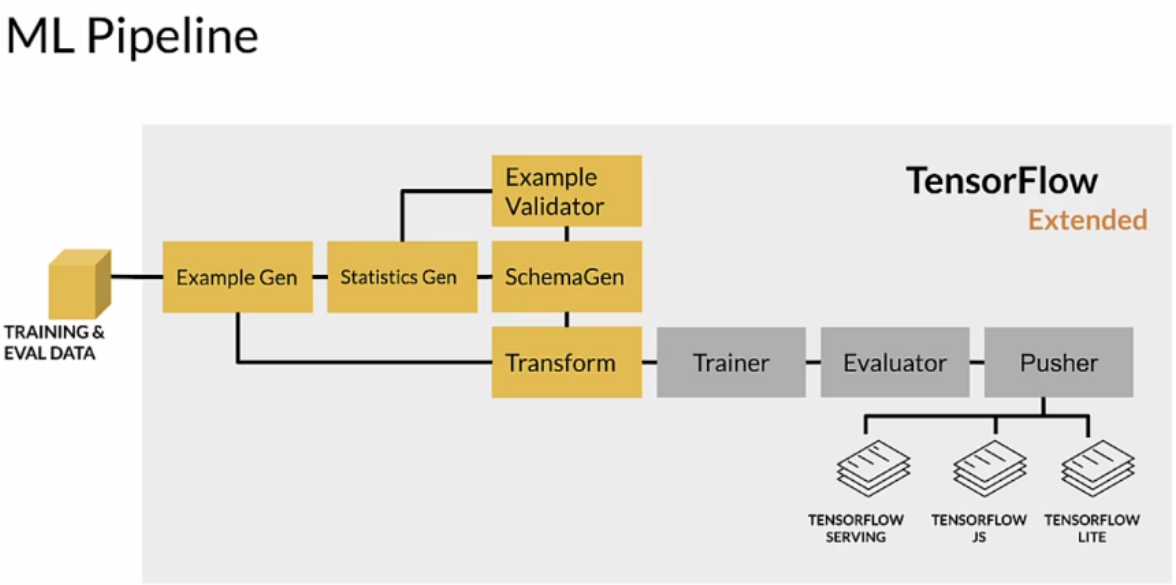
Feature Transformation at Scale

Preprocessing Data at Scale

We were developing our notebooks in python and then when we deployed into storm we had to translate all of the feature engineering that we did into java. Well, as you can imagine, that was not ideal and there were little weird problems that cropped up and were often very difficult to find doing that transformation. This is not an ideal way to do production machine learning.

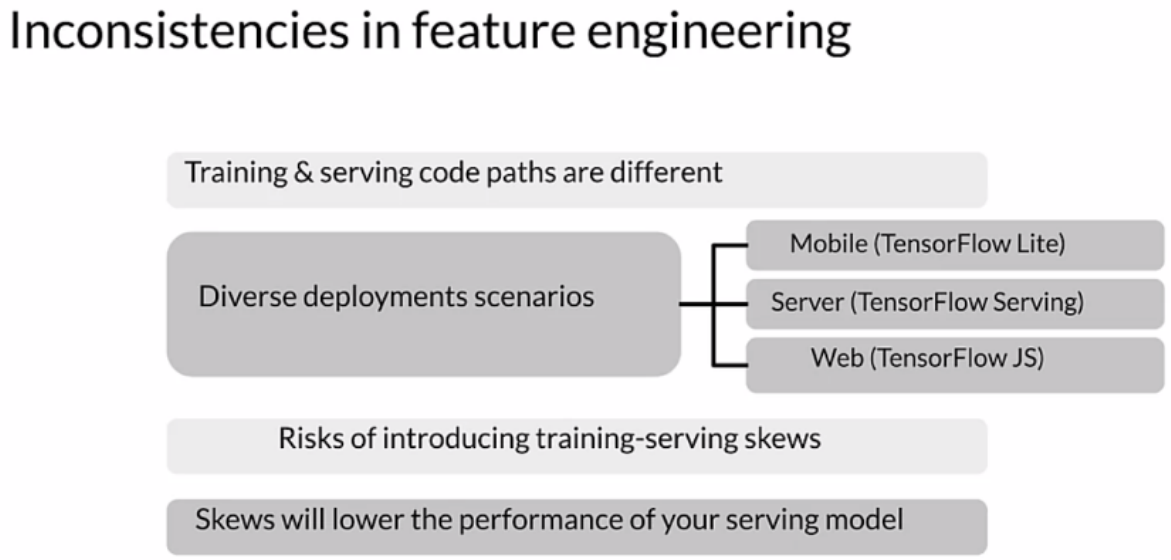


So what is a much better technique is to use a pipeline, a unified framework where you can both train and deploy with consistent and reproducible results.



But there's a number of challenges that you need to deal with. So to preprocessed data at scale, we start with real world models and these could be terabytes of data. So when you're doing this kind of kind of work, you want to start with a subset of your data work out. As many issues as you can think about how that's going to scale up to the terabytes that you need to actually work with your whole dataset. Large scale data processing frameworks are no different than what you're going to use, on your laptop or in a notebook or what have you. So you need to start thinking about that from the beginning of the process, how this is going to be applied and the earlier you can start working with those frameworks. The more you work out the issues early with smaller datasets and quicker turnaround time, consistent transformations between training and serving are incredibly important. If you do different transformations when you're serving your model than

you did when you were training your model, you are going to have problems and often those problems are very hard to find.



So there is a notion of the granularity at which you're doing preprocessing. So you need to think about this, you're going to do transformations, both the instance level and a full pass over your data. And depending on the transformation that you're doing, you may be doing it in one or the other. You can usually always do instance level. But a full path requires that you have the whole dataset. So for clipping, even for clipping, you need to set clipping boundaries. If you're not doing that through some arbitrary setting of those boundaries, if you're using the data set itself to determine how to clip, then you're going to need to do a full pass. So min max for clipping is going to be important. Doing a multiplication at the instance level is fine, but scaling well now I'm going to need probably the standard deviation and maybe the min and max. Bucketizing similarly, unless I know ahead of time what buckets are going to be, I'm going to need to do a full pass to find what buckets make sense. Expanding features I can probably do that at the instance level. So these two things are different at training time. I have the whole dataset so I can do a full pass, although it can be expensive to do that. At serving time, I get individual examples, so I can only really do instance level. So anything I need to do that requires characteristics of the whole dataset. I need to have that saved off so I can use it at serving time.

Preprocessing granularity

Transformations	
Instance-level	**Full-pass**
Clipping	Minimax
Multiplying	Standard scaling
Expanding features	Bucketizing
etc.	etc.

When do you transform?

Pre-processing training dataset

Pros	Cons
Run-once	Transformations reproduced at serving
Compute on entire dataset	Slower iterations

So when do you transform, you're going to pre-process your training dataset and there's pros and cons in how you do that. First of all, you don't you only run **once per training session**, so that's cool. And you're going to compute it on the entire dataset, but only once, for each training session. The cons: well, you're going to have to reproduce all those transformations that serve or save off the information that you learned about the data, like the standard deviation. So that you can use that later while you're serving. And there's, slower iterations around this. Each time you make a change, you've got to make a full pass over the dataset.

So you can do this **within the model**. Transforming within the model has some nice features so there are cons as well. First of all it makes iteration somewhat easier because it's embedded as part of your model and there's guarantees around the transformation that you're doing. But the cons are it can be expensive to do those transforms, especially when your compute resources are limited. And think about when you do a transform within the model, you're going to do that same transform at serving time. So you may have say GPUs or TPUs when you're training, you may not when you're serving. So there's long model latency, that's when you're serving your model, if you're doing a lot of transformation with it that can slow down the response time for your

model and increase latency.And, you can have transformations per batch that skew that we talked about. If you haven't saved those constants that you need, that could be an issue.

How about ‘within’ a model?

Transforming within the model

Pros	Cons
Easy iterations	Expensive transforms
Transformation guarantees	Long model latency
	Transformations per batch: skew

You can also **transform per batch** instead of for the entire dataset. So you could for example, normalize features by their average within the batch. That only requires you to make a pass over each batch and not the full data set, which when you're working with terabytes of data. That can be a significant advantage in terms of processing. And there's ways to normalize per batch as well. So you can compute an average and use that and normalization per batch and then do it again for the next batch there will be differences batch to batch. Sometimes that's good in cases. So for example where you have changes over time in a time series, that can be a good thing but you need to consider that. So normalizing by the average per batch, precomputing the average and using it during normalization, you can use it for multiple batches for example. So this is a different way to think about how to work with your data when you have a large dataset.

Why transform per batch?

- For example, normalizing features by their average
- Access to a single batch of data, not the full dataset
- Ways to normalize per batch
  - Normalize by average within a batch
  - Precompute average and reuse it during normalization

Optimizing instance-level transformations

- Indirectly affect training efficiency
- Typically accelerators sit idle while the CPUs transform
- Solution:
    - Prefetching transforms for better accelerator efficiency

Summarizing the challenges

- Balancing predictive performance
- Full-pass transformations on training data
- Optimizing instance-level transformations for better training efficiency (GPUs, TPUs, ...)

So to **summarize** the challenges we have to balance the predictive performance of our model and the requirements for it. Making full pass transformations on the training data is one thing. If we're going to do that then we need to think about how that works when we serve our model as well and save those constants. And we want to optimize the instance level transformations for better training efficiency. So things like prefetching can really help with that.

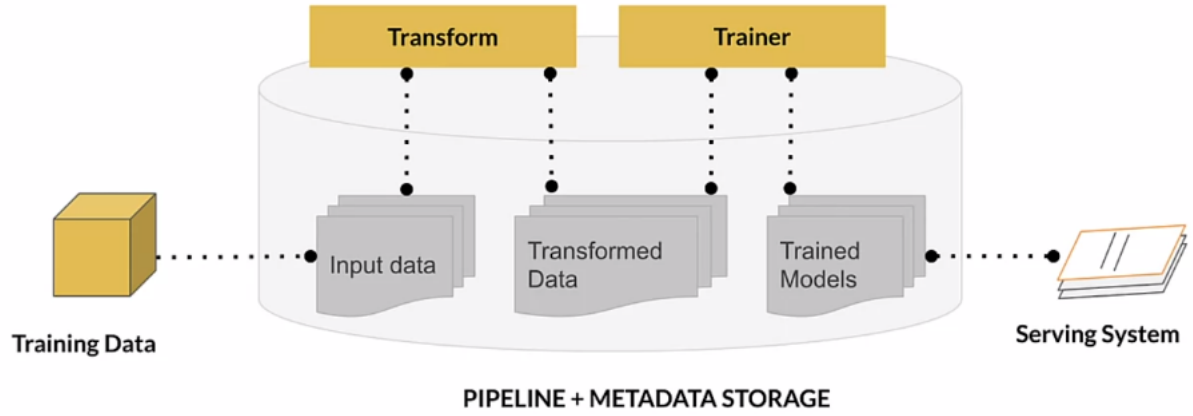
So **key points**, inconsistent data affects the accuracy of the results. So scalable data processing frameworks allow processing of large datasets and distributed inefficient ways. But we also need to think about how that gets applied in serving.

Key points

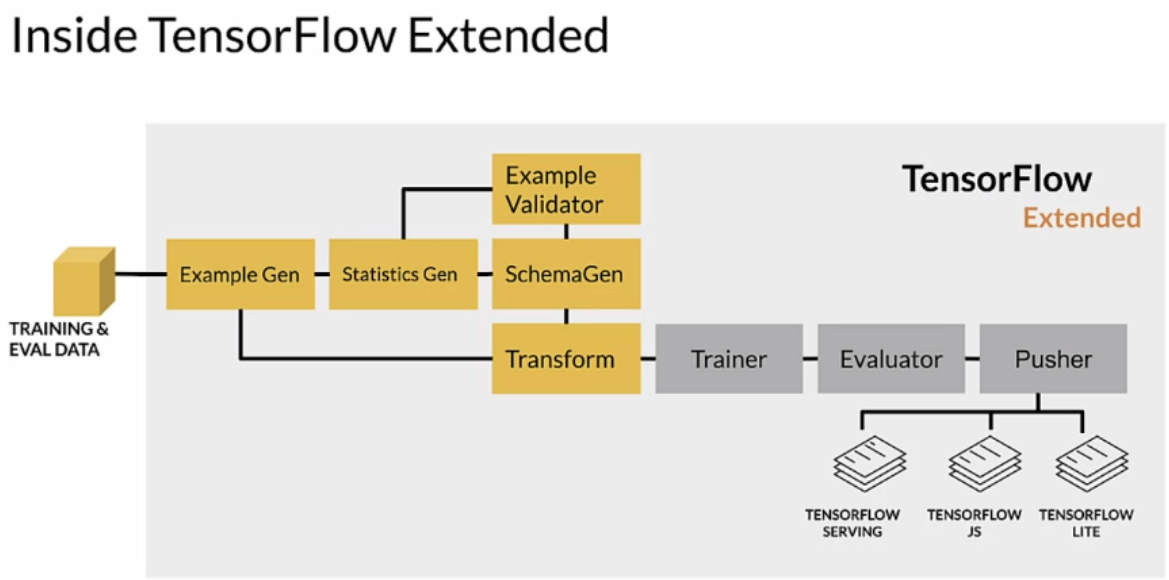
- Inconsistent data affects the accuracy of the results
- Need for scaled data processing frameworks to process large datasets in an efficient and distributed manner

TensorFlow Transform

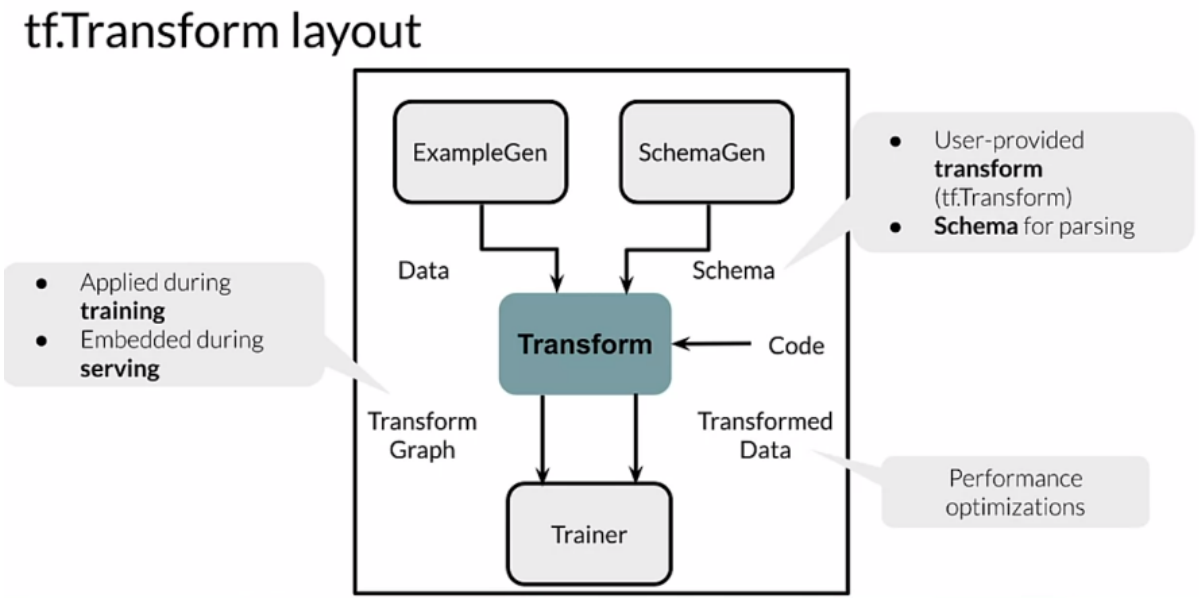
Enter tf.Transform



Looking at this a little differently, we're starting with our training and eval data. We've actually split our dataset. We split it with **ExampleGen**. Those get fed to **StatisticsGen**. These are both TFX components within a TFX pipeline. StatisticsGen calculates statistics for our data. For each of the features, if they're numeric features, for example, what is the mean of that feature value? What is the standard deviation? The min, the max, so forth. Those statistics get fed to **SchemaGen** which infers the types of each of our features. That creates a schema that is then used by downstream components including **Example Validator**, which takes those statistics and that schema, and it looks for problems in our data. We won't go into great detail here about the things that it looks for, but if we have examples that are the wrong type in a particular feature, so maybe we have an integer where we expected a float. Now, we're getting into **Transform**. Transform is going to take the schema that was generated on the original split dataset, and it's going to do our feature engineering. Transform is where the feature engineering happens. That gets given the **Trainer**, there's **Evaluator** that evaluates the results, a set of **Pusher** that pushes it to our deployment targets which is, however we're serving our models, so **TENSORFLOW SERVING**, or **JS**, or **LITE**, wherever it is that we're serving our model.

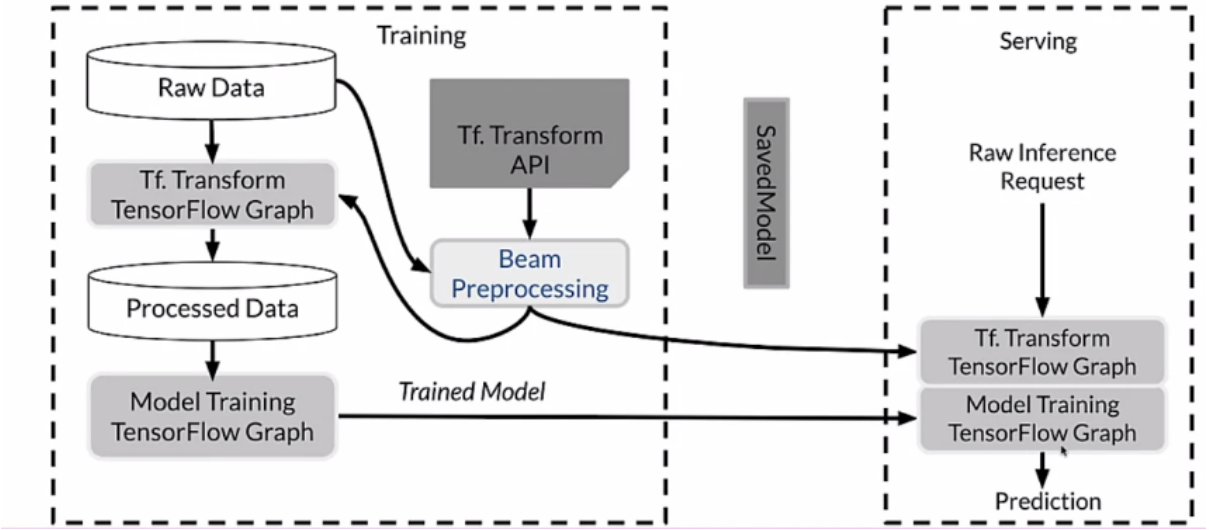


Internally, if we want to look at the transform component, it's getting inputs from, as we saw, ExampleGen and SchemaGen. That is the data that was originally split by Example Gen, and the schema that was generated by SchemaGen. That schema, by the way, may very well have been reviewed and improved by a developer who knew more about what to expect from the data than can be really inferred by SchemaGen. That's referred to as curating the schema. Transform gets that and it also gets a lot of user code because we need to express the feature engineering we want to do. If we're going to normalize a feature, we need to give it user code to tell transform to do that. The result is a TensorFlow graph, which was referred to as the transform graph and the transform data itself. The graph expresses all of the transformations that we are doing on our data, as a TensorFlow graph. The transform data is simply the result of doing all those transformations. Those are given to Trainer, which is going to use the transformed data for training and it's going to include the transform graph and we'll take a look at that in a second. We have a user provided transform component and we have a schema for it. We apply our transformations during training and as we'll see later, we also apply those transformations at serving time. There's a lot of performance optimizations that we apply as well, and we'll take a look at that in a second.

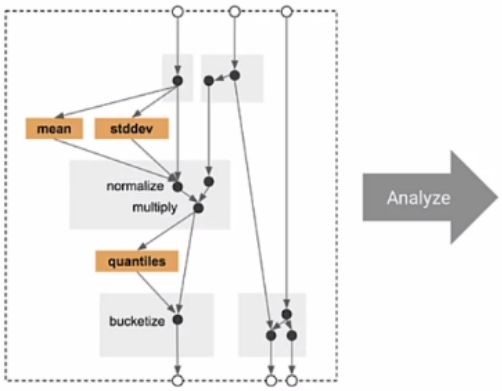


Continuing a little deeper here, we have our raw data and we have our transform component. It produces a TensorFlow graph. The result of running that graph is processed data, it's been transformed. That gets given in model training, as processed data is given to our trainer component, where we use it for training our model. Training our model creates a TensorFlow graph. Notice now we have two different graphs. We have a graph here from Transform and a graph here from Training. Using the Tf Transform API, we express the feature engineering that we want to do and we give that code, or rather the transform component gives that code to a Apache Beam distributed processing cluster. That way we can do, and remember this is all designed to work with potentially terabytes of data. That could be quite a bit of processing to do all of that transformation using a distributed processing cluster by using Apache Beam gives us the capacity to do that. The result is a saved model. Saved model is just a format that expresses a trained model as a saved model. That gets included, we have both the transform graph and the training graph. Those are given to our serving infrastructure. Now we have both of those graphs, the feature engineering that we did, and the model itself, the trained model and those are used when we serve requests.

tf. Transform: Going deeper



tf.Transform Analyzers



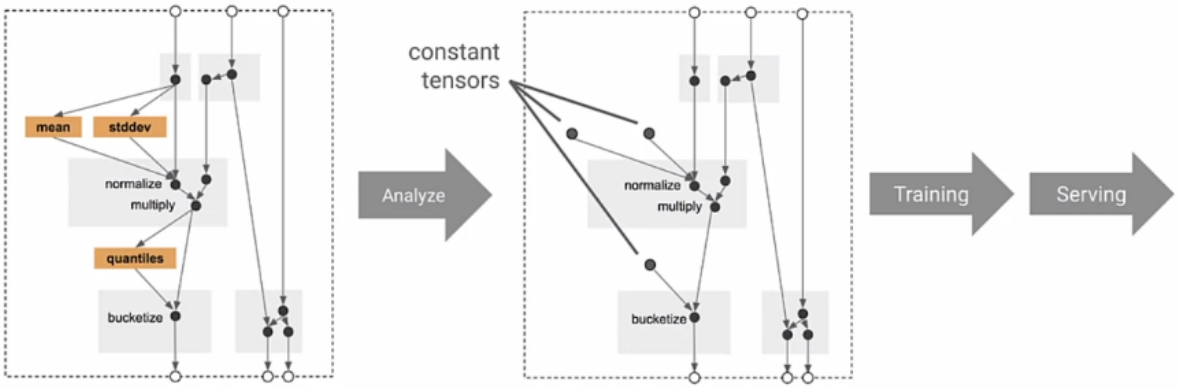
They behave like TensorFlow Ops, but run only once during training

For example:
tft.min *computes the minimum of a tensor over the training dataset*

Looking at this a little differently, we have analyzers. Again, we're using TensorFlow and we're using the TensorFlow Transform API. We're using TensorFlow Ops and a big part of what they do, what an Analyzer does is it makes a full pass over our dataset in order to collect constants that we're going to need when we do feature engineering. For example, if we're going to do a min-max, well, we need to make a full pass through our data to know what the min and the max are for each feature that we're using for that. Those are constants that we need to express. We're going to use an Analyzer to make that pass over the data and collect those constants. It's also going to express the operations that we're going to do. They behave like TensorFlow Ops, but they only were on once during training and then they're saved off as a graph. For example, if we're using the tft.min, which is one of the methods in the transform STK, it'll compute the minimum of a tensor over the training dataset.

Looking at how that gets applied, we have the graph that is our feature engineering. We run an analysis across our dataset to collect the constants that we need. Really what we're doing is we're enabling us later to apply these into our Transform graph, so that we can transform individual examples without making a pass over our entire dataset. That gets applied during training, and the same exact graph gets applied during serving. There is no potential here for training and serving skew or having different code paths when we train our model versus when we serve our model that causes problems and those not be equivalent. We're using exactly the same graph in both places, so there is no potential for doing that.

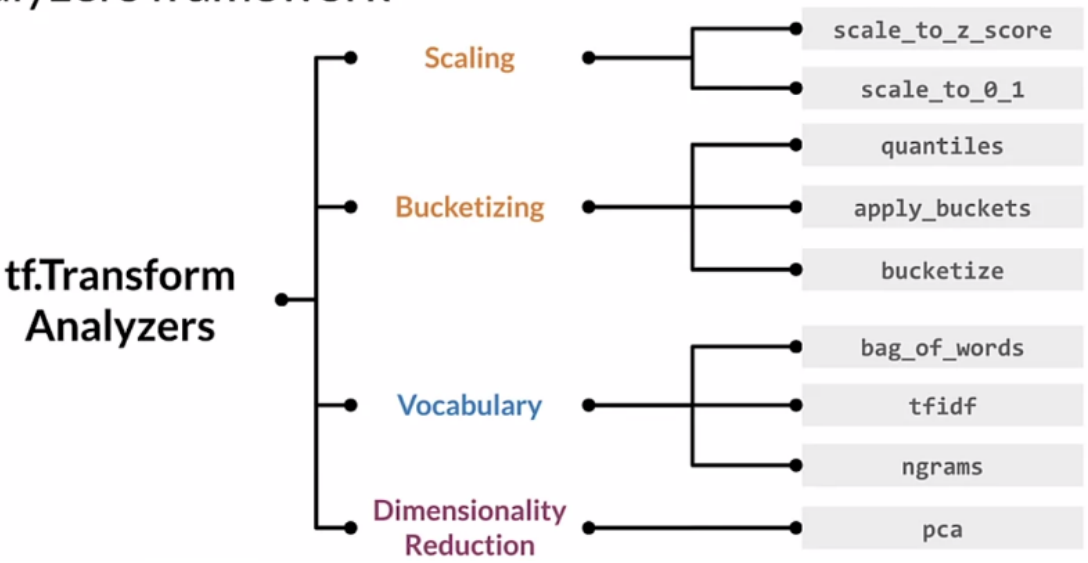
How Transform applies feature transformations



Benefits of using tf.Transform

- Emitted tf.Graph holds all necessary constants and transformations
- Focus on data preprocessing only at training time
- Works in-line during both training and serving
- No need for preprocessing code at serving time
- Consistently applied transformations irrespective of deployment platform

Analyzers framework



tf.Transform preprocessing_fn

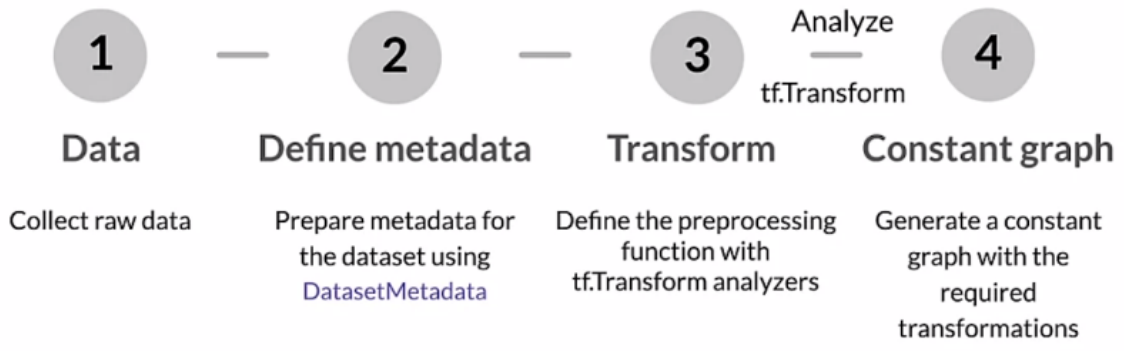
```
def preprocessing_fn(inputs):
    ...

for key in DENSE_FLOAT_FEATURE_KEYS:
    outputs[key] = tft.scale_to_z_score(inputs[key])

for key in VOCAB_FEATURE_KEYS:
    outputs[key] = tft.vocabulary(inputs[key], vocab_filename=key)

for key in BUCKET_FEATURE_KEYS:
  outputs[key] = tft.bucketize(inputs[key], FEATURE_BUCKET_COUNT)
```

Commonly used imports

```
import tensorflow as tf
import apache_beam as beam
import apache_beam.io.iobase

import tensorflow_transform as tft
import tensorflow_transform.beam as tft_beam
```

Hello World with tf.Transform

Hello world with tf.Transform



Collect raw samples (Data)

```
[
   {'x': 1, 'y': 1, 's': 'hello'},
   {'x': 2, 'y': 2, 's': 'world'},
   {'x': 3, 'y': 3, 's': 'hello'}
]
```

Inspect data and prepare metadata (Data)

```
from tensorflow_transform.tf_metadata import (
    dataset_metadata, dataset_schema)

raw_data_metadata = dataset_metadata.DatasetMetadata(
    dataset_schema.from_feature_spec({

    'y': tf.io.FixedLenFeature([], tf.float32),
        'x': tf.io.FixedLenFeature([], tf.float32),
        's': tf.io.FixedLenFeature([], tf.string)
    }))
```

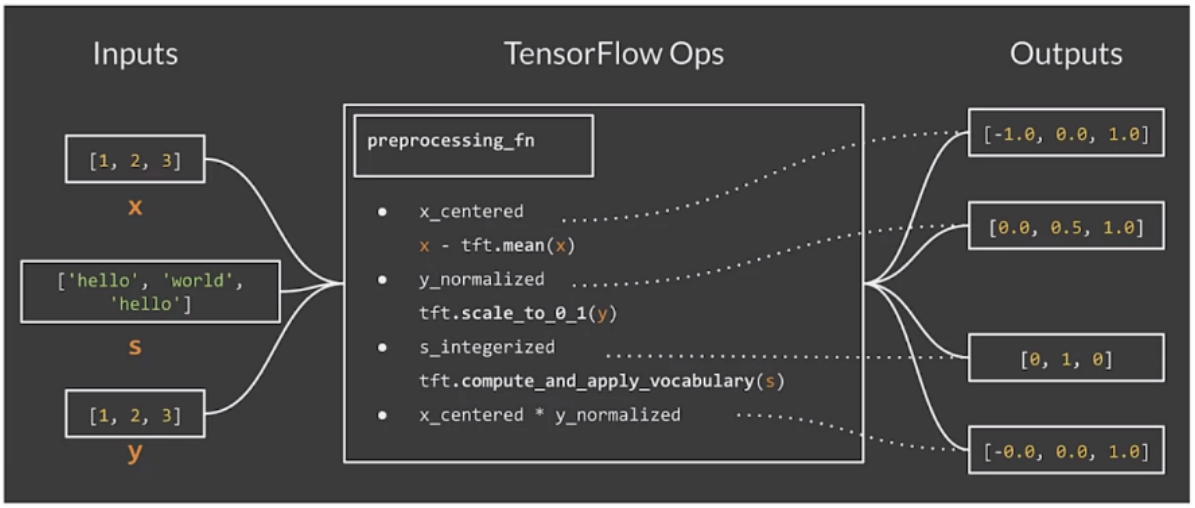
Preprocessing data (Transform)

```
def preprocessing_fn(inputs):
"""Preprocess input columns into transformed columns."""
    x, y, s = inputs['x'], inputs['y'], inputs['s']
    x_centered = x - tft.mean(x)
    y_normalized = tft.scale_to_0_1(y)
    s_integerized = tft.compute_and_apply_vocabulary(s)
    x_centered_times_y_normalized = (x_centered * y_normalized)
```

Preprocessing data (Transform)

```
    return {
      'x_centered': x_centered,
     'y_normalized': y_normalized,
     's_integerized': s_integerized,
     'x_centered_times_y_normalized': x_centered_times_y_normalized,
    }
```

Tensors in… tensors out



Running the pipeline

```
def main():
with tft_beam.Context(temp_dir=tempfile.mkdtemp()):
  transformed_dataset, transform_fn = (
      (raw_data, raw_data_metadata) | tft_beam.AnalyzeAndTransformDataset(
          preprocessing_fn))
```

Running the pipeline

```
  transformed_data, transformed_metadata = transformed_dataset



  print('\nRaw data:\n{}\n'.format(pprint.pformat(raw_data)))
  print('Transformed data:\n{}'.format(pprint.pformat(transformed_data)))

if __name__ == '__main__':
  main()
```

Before transforming with tf.Transform

```
    # Raw data:
    [{'s': 'hello', 'x': 1, 'y': 1},
     {'s': 'world', 'x': 2, 'y': 2},
     {'s': 'hello', 'x': 3, 'y': 3}]
```

After transforming with tf.Transform

```
  # After transform
  [{'s_integerized': 0,
    'x_centered': -1.0,
    'x_centered_times_y_normalized': -0.0,
    'y_normalized': 0.0},
   {'s_integerized': 1,
    'x_centered': 0.0,
    'x_centered_times_y_normalized': 0.0,
    'y_normalized': 0.5},
   {'s_integerized': 0,
    'x_centered': 1.0,
    'x_centered_times_y_normalized': 1.0,
    'y_normalized': 1.0}]
```

Key points

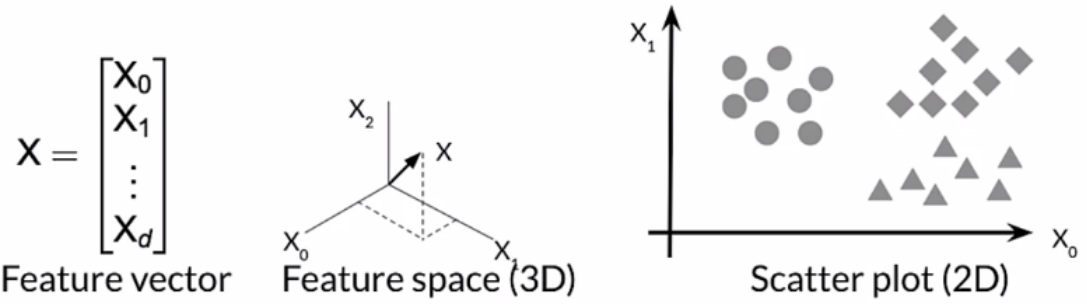
- tf.Transform allows the pre-processing of input data and creating features
- tf.Transform allows defining pre-processing pipelines and their execution using large-scale data processing frameworks
- In a TFX pipeline, the Transform component implements feature engineering using TensorFlow Transform

Feature Selection

Feature Spaces

Feature space

- N dimensional space defined by your N features
- Not including the target label



Let's see an example of house pricing.

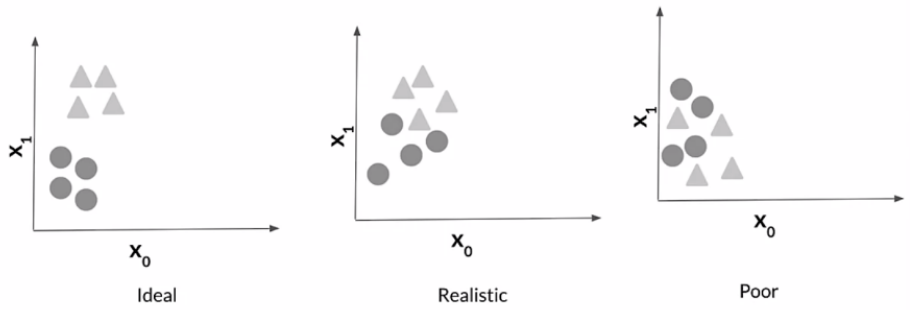
3D Feature Space

No. of Rooms X_0	Area X_1	Locality X_2	Price Y
5	1200 sq. ft	New York	$40,000
6	1800 sq. ft	Texas	$30,000

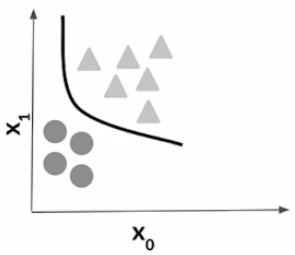
$Y = f(X_0, X_1, X_2)$

f is your ML model acting on feature space X_0, X_1, X_2

2D Feature space - Classification



Drawing decision boundary



Model learns decision boundary

Boundary used to classify data points

Feature space coverage

- Train/Eval datasets representative of the serving dataset
  - Same numerical ranges
  - Same classes
  - Similar characteristics for image data
  - Similar vocabulary, syntax, and semantics for NLP data

Ensure feature space coverage

- Data affected by: seasonality, trend, drift.
- Serving data: new values in features and labels.
- Continuous monitoring, key for success!

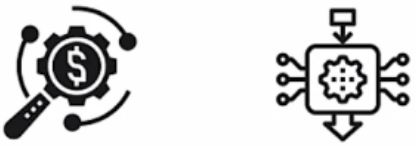
Feature Selection

Feature selection

- Identify features that best represent the relationship
- Remove features that don't influence the outcome
- Reduce the size of the feature space
- Reduce the resource requirements and model complexity

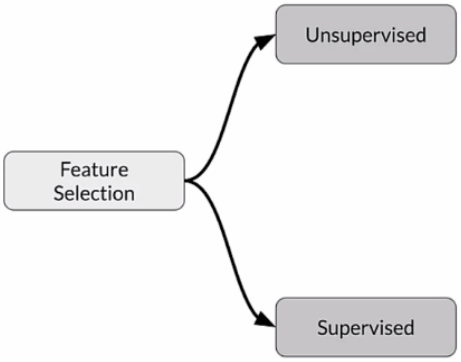
Why is feature selection needed?

Reduce storage and I/O requirements

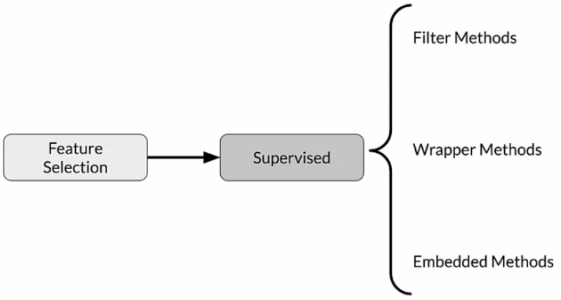


Minimize training and inference costs

Feature selection methods



Supervised methods



Unsupervised feature selection

1. **Unsupervised**

- Features-target variable relationship not considered
- Removes redundant features (correlation)

Supervised feature selection

2. Supervised

- Uses features-target variable relationship
- Selects those contributing the most

Practical example

Feature selection techniques on Breast Cancer Dataset (Diagnostic)

Predicting whether tumour is benign or malignant.

Feature list

id	diagnosis	radius-mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
842302	M	17.99	10.38	122.8	1001.0	0.1184	0.2776
concavity_mean	concavepoints_mean	symmetry_mean	fractal_dimension_mean	radius_se	texture_se	perimeter_se	area_se
0.3001	0.1471	0.2419	0.07871	1.095	0.9053	8.589	153.4
smoothness_se	compactness_se	concavity_se	concavepoints_se	symmetry_se	fractal_dimension_se	radius-worst	texture_worst
0.0064	0.049	0.054	0.016	0.03	0.006	25.38	17.33
perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concavepoints_worst	symmetry_worst	fractal_dimension_worst	Unnamed:32
184.6	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.1189	NaN

Irrelevant features

Performance evaluation

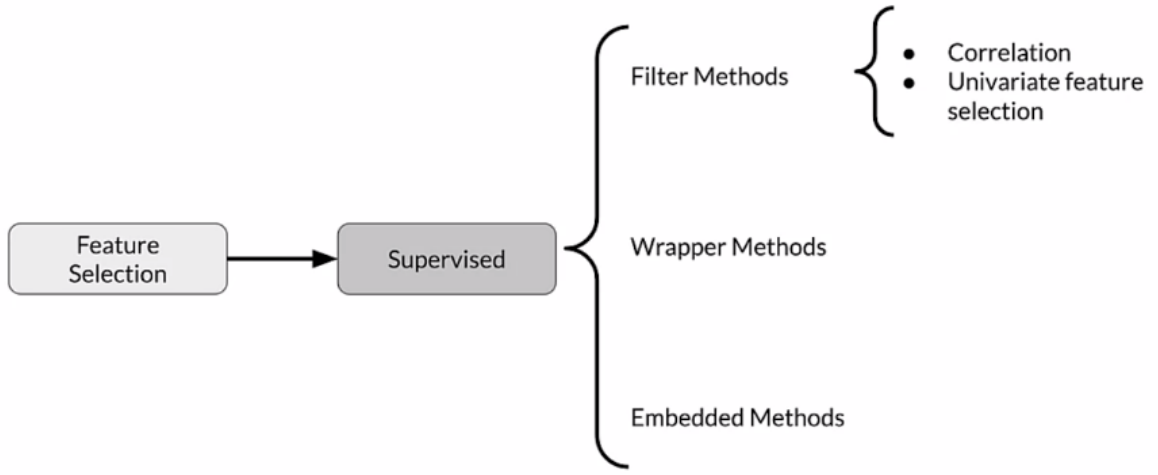
We train a **RandomForestClassifier** model in sklearn.ensemble on selected features

Metrics (`sklearn.metrics`):

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488

Filter Methods

Filter methods

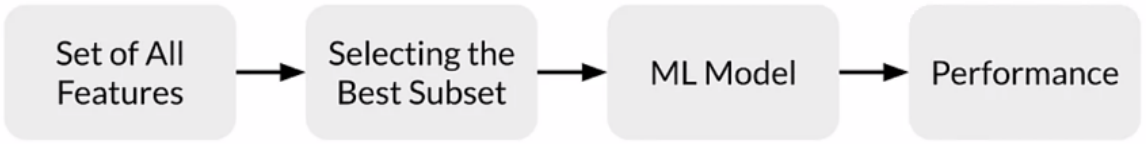


Filter methods

- Correlated features are usually redundant
  - Remove them!

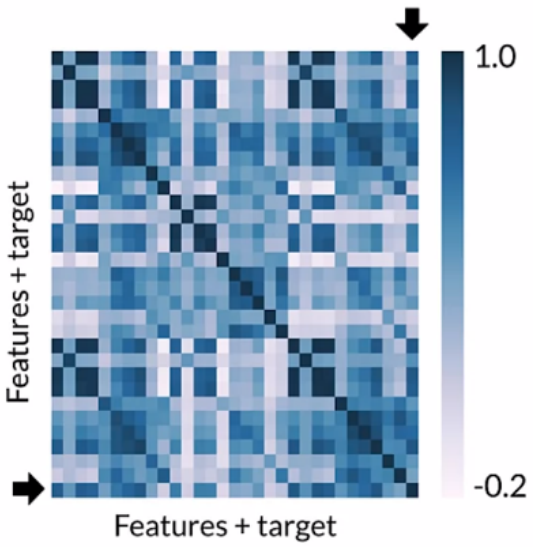
Popular filter methods:

- Pearson Correlation
  - Between features, and between the features and the label
- Univariate Feature Selection



Correlation matrix

- Shows how features are related:
  - To each other (Bad)
  - And with target variable (Good)
- Falls in the range [-1, 1]
  - 1 High positive correlation
  - -1 High negative correlation



Feature comparison statistical tests

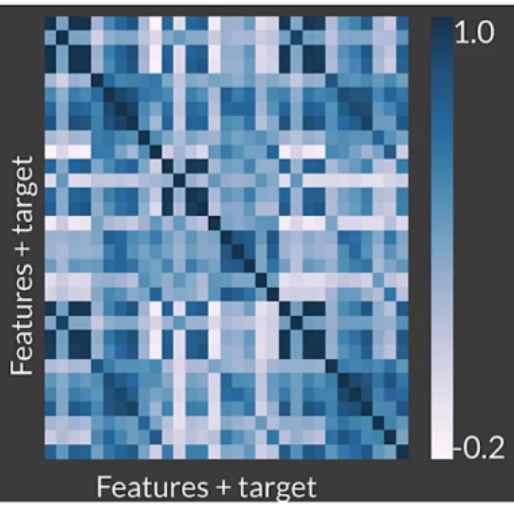
- Pearson's correlation: Linear relationships
- Kendall Tau Rank Correlation Coefficient: Monotonic relationships & small sample size
- Spearman's Rank Correlation Coefficient: Monotonic relationships

Other methods:

- Mutual information
- F-Test
- Chi-Squared test

Determine correlation

```
# Pearson's correlation by default
cor = df.corr()


plt.figure(figsize=(20,20))
# Seaborn
sns.heatmap(cor, annot=True, cmap=plt.cm.PuBu)
plt.show()
```



Selecting features

```
cor_target = abs(cor["diagnosis_int"])

# Selecting highly correlated features as potential features to eliminate
relevant_features = cor_target[cor_target>0.2]
```

Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	**21**	**0.974206**	**0.973684**	**0.953488**	**0.97619**	**0.964706**

Univariate feature selection in SKLearn

SKLearn Univariate feature selection routines:

1. **`SelectKBest`**
2. `SelectPercentile`
3. `GenericUnivariateSelect`

Statistical tests available:

- Regression: `f_regression`, `mutual_info_regression`
- Classification: `chi2`, `f_classif`, `mutual_info_classif`

There is a bug in standardizing `X_test` here and in some of the next videos. It should be transformed using the parameters computed from `X_train` . You will see this correction in the ungraded lab at the end of this lesson.

SelectKBest implementation

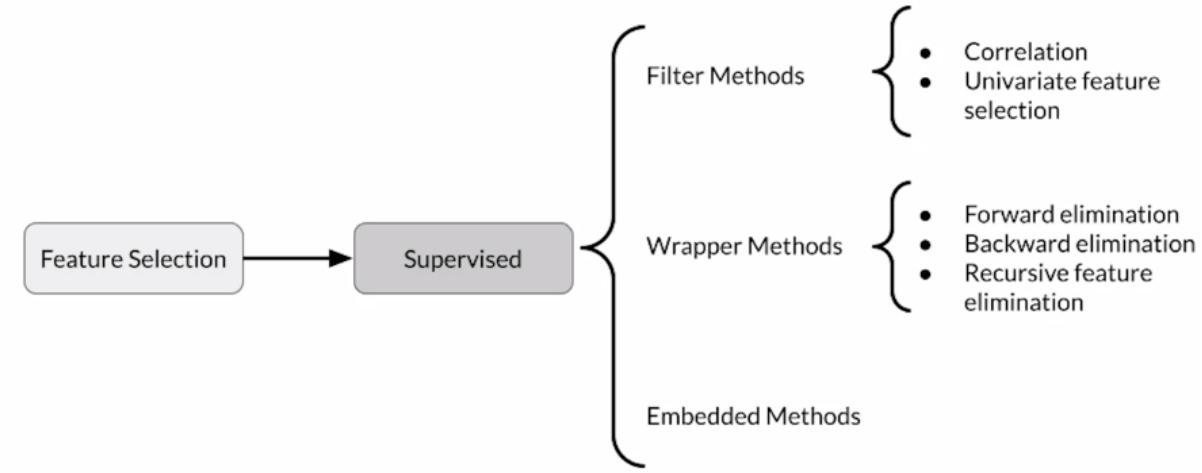
```
def univariate_selection():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                          test_size = 0.2,stratify=Y, random_state = 123)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  min_max_scaler = MinMaxScaler()
  Scaled_X = min_max_scaler.fit_transform(X_train_scaled)
  selector = SelectKBest(chi2, k=20) # Use Chi-Squared test
  X_new = selector.fit_transform(Scaled_X, Y_train)
  feature_idx = selector.get_support()
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[feature_idx]
  return feature_names
```

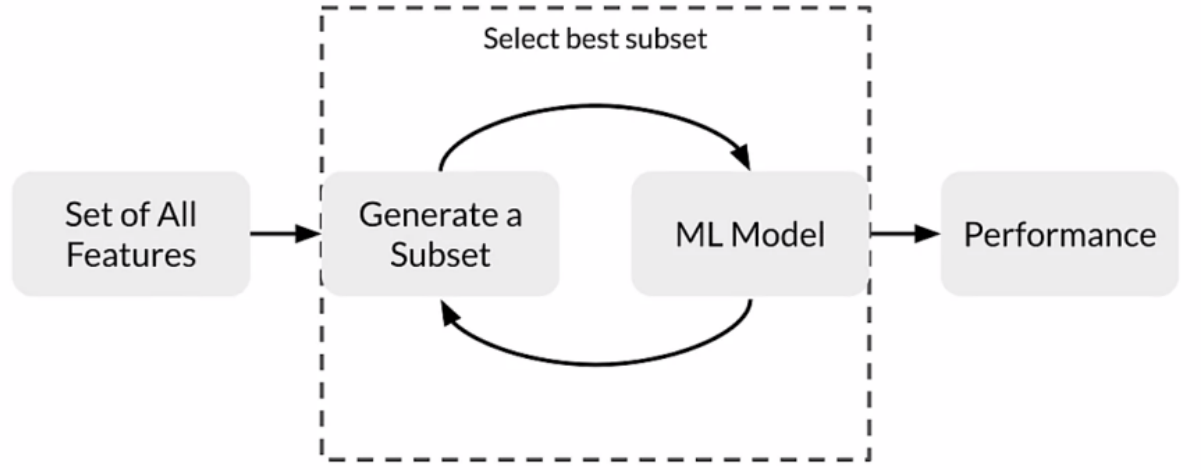
Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.967262	0.964912	0.931818	0.97619	0.953488
Correlation	21	0.974206	0.973684	0.953488	0.97619	0.964706
Univariate (Chi^2)	**20**	**0.960317**	**0.95614**	**0.91111**	**0.97619**	**0.94252**

Wrapper Methods

Wrapper methods





Forward selection

1. Iterative, greedy method
2. Starts with 1 feature
3. Evaluate model performance when **adding** each of the additional features, one at a time
4. Add next feature that gives the best performance
5. Repeat until there is no improvement

Backward elimination

1. Start with all features
2. Evaluate model performance when **removing** each of the included features, one at a time
3. Remove next feature that gives the best performance
4. Repeat until there is no improvement

Recursive feature elimination (RFE)

1. Select a model to use for evaluating feature importance
2. Select the desired number of features
3. Fit the model
4. Rank features by importance
5. Discard least important features
6. Repeat until the desired number of features remains

Recursive feature elimination

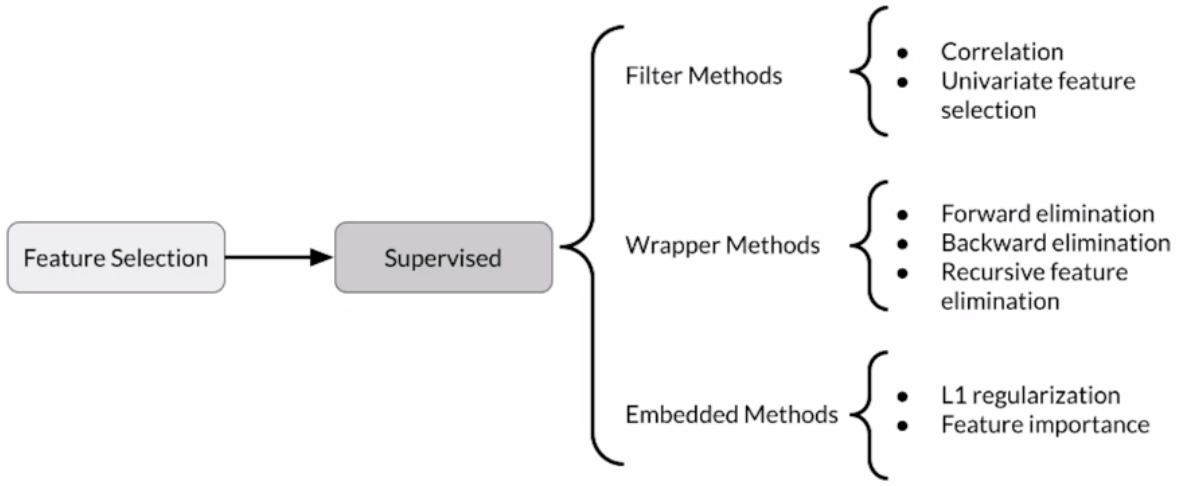
```
def run_rfe():

  X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size = 0.2, random_state = 0)

  X_train_scaled = StandardScaler().fit_transform(X_train)
  X_test_scaled = StandardScaler().fit_transform(X_test)

  model = RandomForestClassifier(criterion='entropy', random_state=47)
  rfe = RFE(model, 20)
  rfe = rfe.fit(X_train_scaled, y_train)
  feature_names = df.drop("diagnosis_int",axis = 1 ).columns[rfe.get_support()]
  return feature_names

rfe_feature_names = run_rfe()

rfe_eval_df = evaluate_model_on_features(df[rfe_feature_names], Y)
rfe_eval_df.head()
```

Performance table

Method	Feature Count	Accuracy	AUROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.96491	0.931818	0.97619	0.953488
Correlation	21	0.97420	0.97368	0.9534883	0.97619	0.964705
Univariate (Chi^2)	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	**20**	**0.97420**	**0.97368**	**0.953488**	**0.97619**	**0.964706**

Embedded Methods

Embedded methods



Feature importance

- Assigns scores for each feature in data
- Discard features scored lower by feature importance

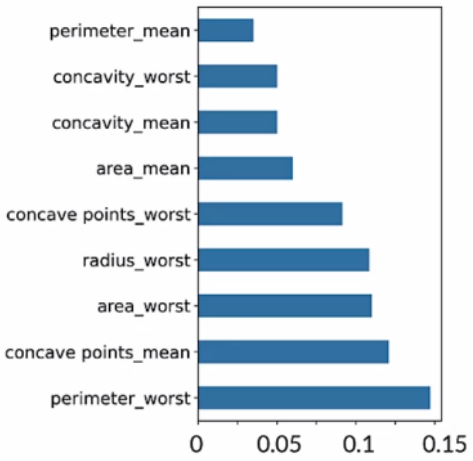
Feature importance with SKLearn

- Feature Importance class is in-built in Tree Based Models (eg., `RandomForestClassifier`)
- Feature importance is available as a property `feature_importances_`
- *We can then use* `SelectFromModel` *to select features from the trained model based on assigned feature importances.*

Extracting feature importance

```
def feature_importances_from_tree_based_model_():

  X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.2,
                                                      stratify=Y, random_state = 123)
  model = RandomForestClassifier()
  model = model.fit(X_train,Y_train)

  feat_importances = pd.Series(model.feature_importances_, index=X.columns)
  feat_importances.nlargest(10).plot(kind='barh')
  plt.show()

  return model
```

Feature importance plot



Select features based on importance

```
def select_features_from_model(model):


  model = SelectFromModel(model, prefit=True, threshold=0.012)

  feature_idx = model.get_support()
  feature_names = df.drop("diagnosis_int",1 ).columns[feature_idx]
  return feature_names
```

Tying together and evaluation

```
# Calculate and plot feature importances
model = feature_importances_from_tree_based_model_()

# Select features based on feature importances
feature_imp_feature_names = select_features_from_model(model)
```

Performance table

Method	Feature Count	Accuracy	ROC	Precision	Recall	F1 Score
All Features	30	0.96726	0.964912	0.931818	0.9761900	0.953488
Correlation	21	0.97420	0.973684	0.953488	0.9761904	0.964705
Univariate Feature Selection	20	0.96031	0.95614	0.91111	0.97619	0.94252
Recursive Feature Elimination	20	0.9742	0.973684	0.953488	0.97619	0.964706
Feature Importance	14	0.96726	0.96491	0.931818	0.97619	0.953488

Embedded methods combine the best of both worlds, filter and wrapper methods:

- Faster than wrapper methods: Wrapper methods are based on the greedy algorithm and thus solutions are slow to compute.
- More efficient than filter methods: Filter methods suffer from inefficiencies as they need to look at all the possible feature subsets.

Week 2 Optional References

Week 2: Feature Engineering, Transformation and Selection

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Mapping raw data into feature

Feature engineering techniques

Scaling

Facets

Embedding projector

Encoding features

TFX:

1. https://www.tensorflow.org/tfx/guide#tfx_pipelines
2. https://ai.googleblog.com/2017/02/preprocessing-for-machine-learning-with.html

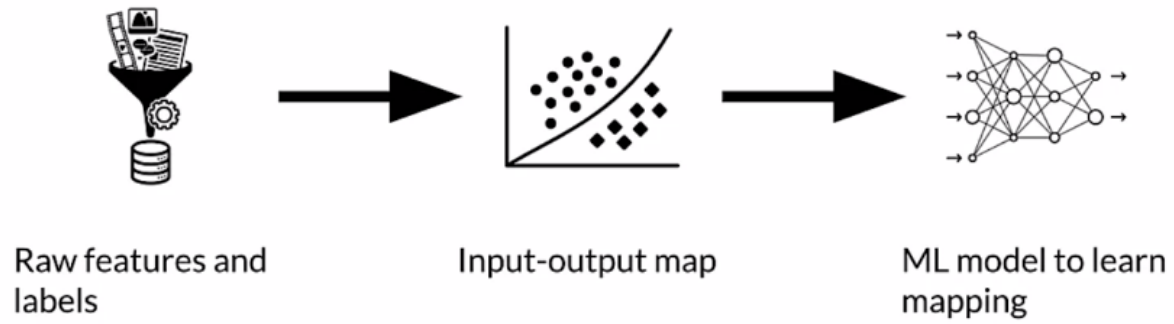
Breast Cancer Dataset

Week 3: Data Journey and Data Storage

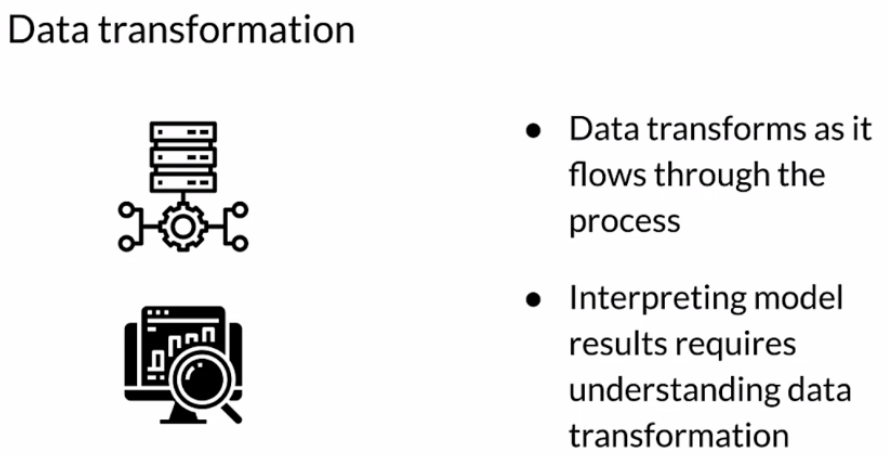
Data Journey and Data Storage

Data Journey

Trying to understand data provenance begins with the Data journey. The journey starts with raw features and labels from whatever sources that we have. The Data describes a function that maps the input in the training set to the labels that we're trying to predict. During training, the model learns the functional mapping from the input to the labels so as to be as accurate as possible. The Data transforms and changes as part of this training process.



The Data transforms and changes as part of this training process. As the Data flows through the process it transforms. Examples of this are, are changing data formats, applying feature engineering and training the model to make predictions. There's a dual connection between understanding these data transformations and interpreting the model's results. Therefore, it's important to track and document these changes closely.



Data artifacts are created as pipeline components execute. What exactly is an artifact? Each time a component produces a result, it generates an artifact. This includes basically everything that is produced by the pipeline, including the data in different stages of transformation, often as a result of feature engineering and the model itself and things like the schema, and metrics and so forth. Basically, everything that's produced, every result that is produced as an artifact.

Artifacts and the ML pipeline

- Artifacts are created as the components of the ML pipeline execute
- Artifacts include all of the data and objects which are produced by the pipeline components
- This includes the data, in different stages of transformation, the schema, the model itself, metrics, etc.

Data provenance and lineage

- The chain of transformations that led to the creation of a particular artifact.
- Important for debugging and reproducibility.

Data provenance: Why it matters

Helps with debugging and understanding the ML pipeline:

Inspect artifacts at each point in the training process

Trace back through a training run

Compare training runs

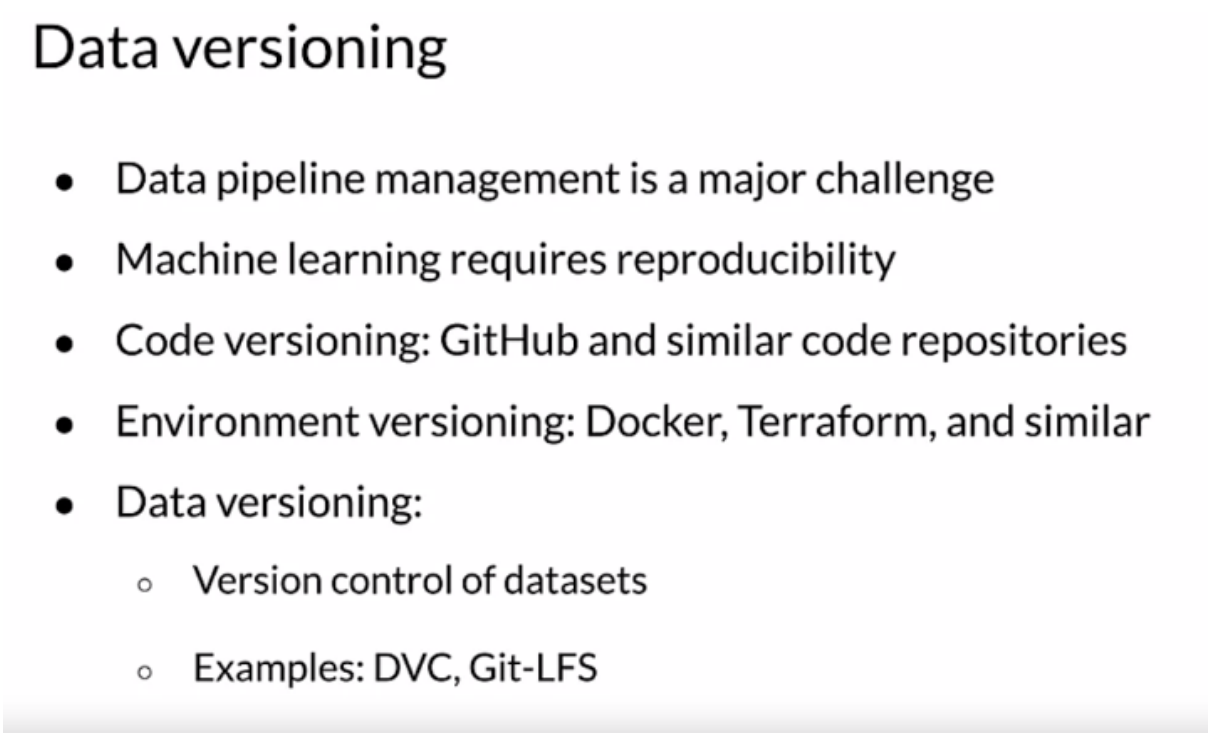
Data lineage: data protection regulation

- Organizations must closely track and organize personal data
- Data lineage is extremely important for regulatory compliance

Data provenance: Interpreting results

Data transformations sequence leading to predictions

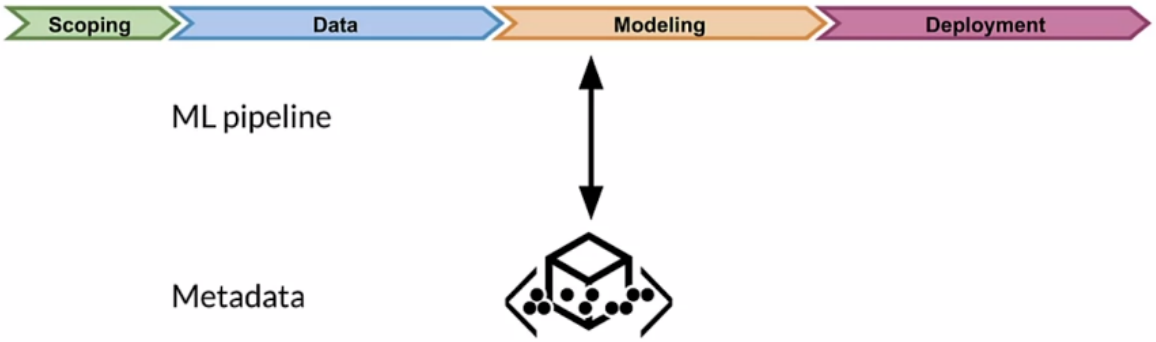
Understanding the model as it evolves through runs



Introduction to ML Metadata

Now let's start exploring how ML metadata or MLMD can help you with tracking artifacts and pipeline changes during a production life cycle. Every run of a production ML pipeline generates metadata containing information about the various pipeline components and their executions or training runs and the resulting artifacts. For example, trained models. In the event of unexpected pipeline behavior or errors, this metadata can be leveraged to analyze the lineage of pipeline components and to help you debug issues. Think of this metadata as the equivalent of logging in software development. MLMD helps you understand and analyze all the interconnected parts of your ML pipeline, instead of analyzing them in isolation.

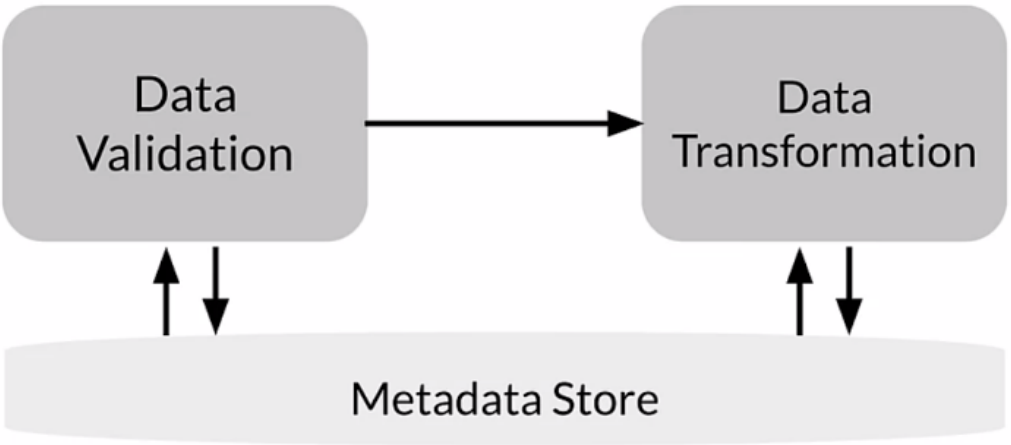
Metadata: Tracking artifacts and pipeline changes



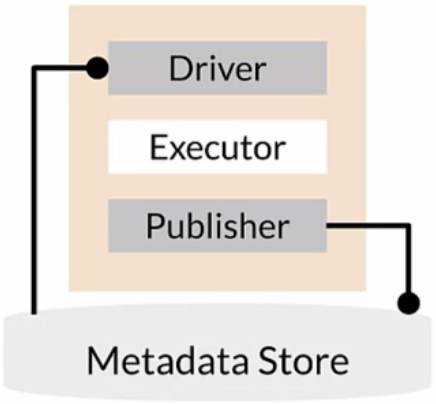
Now consider the two stages in ML engineering that you've seen so far. First you've done data validation, then you've passed the results onto data transformation or feature engineering. This is the first part of any model training process.

But what if you had a centralized repository where every time you run a component, you store the result or the update or any other output of that stage into a repository. So whenever any changes are made, which causes a different result, you don't need to worry about the progress you've made so far getting lost. You can examine your previous results to try to understand what happened and make corrections or take advantage of improvements.

Metadata: Tracking progress



Metadata: TFX component architecture



- Driver:
  - Supplies required metadata to executor
- Executor:
  - Place to code the functionality of component
- Publisher:
  - Stores result into metadata

Let's take a closer look. In addition to the executor where your code runs, each component also includes two additional parts, the **driver** and **publisher**. The **executor** is where the work of the component is done and that's what makes different components different. Whatever input is needed for the executor, is provided by the driver, which gets it from the metadata store. Finally, the publisher will push the results of running the executor back into the metadata store. Most of the time, you won't need to customize the driver or publisher. Creating custom components is almost always done by creating a custom executor.

ML Metadata library

- Tracks metadata flowing between components in pipeline
- Supports multiple storage backends

Now, let's look specifically at ML Metadata or MLMD. MLMD is a library for tracking and retrieving metadata associated with ML developer and data scientist workflows. MLMD can be used as an integral part of an ML pipeline or it can be used independently. However, when integrated with an ML pipeline, you may not even explicitly interact with MLMD. Objects which are stored in MLMD are referred to as artifacts. MLMD stores the properties of each artifact in a relational database and stores large objects like data sets on disc or in a file system or block store.

When you're working with ML metadata, you need to know how data flows between different successive components. Each step in this data flow is described through an entity that you need to be familiar with. At the highest level of MLMD, there are some data entities that can be considered as **units**.

First, there are artifacts. An artifact is an elementary unit of data that gets fed into the ML metadata store and the data is consumed as input or generated as output of each component.

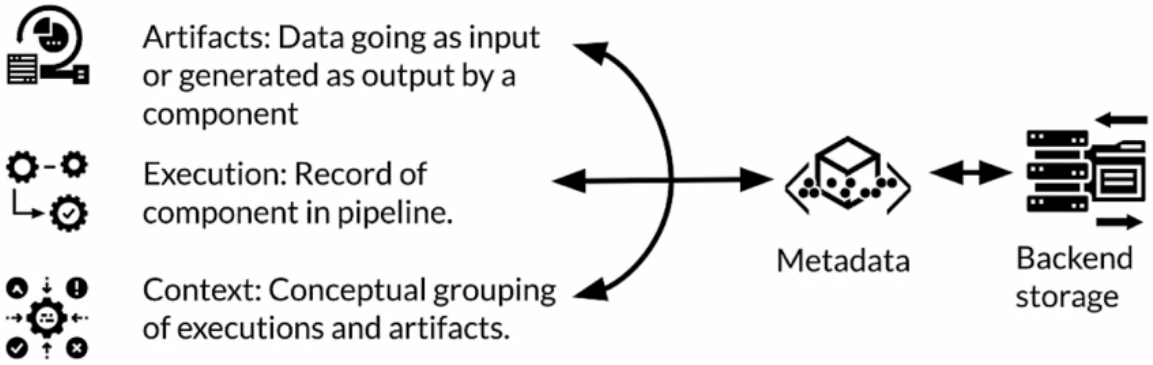
Next there are executions. Each execution is a record of any component run during the ML pipeline workflow, along with its associated runtime parameters. Any artifact or execution will be associated with only one type of component. Artifacts and executions can be clustered together for each type of component separately. This grouping is referred to as the context. A context may hold the metadata of the projects being run, experiments being conducted, details about pipelines, etc. Each of these units can hold additional data describing it in more detail using properties. Next there are **types**, previously, you've seen several types of units that get stored inside the ML metadata. Each type includes the properties of that type. Lastly, we have **relationships**. Relationships store the various units getting generated or consumed when interacting with other units. For example, an event is the record of a relationship between an artifact and an execution.

ML Metadata terminology

Units	Types	Relationships
Artifact	ArtifactType	Event
Execution	ExecutionType	Attribution
Context	ContextType	Association

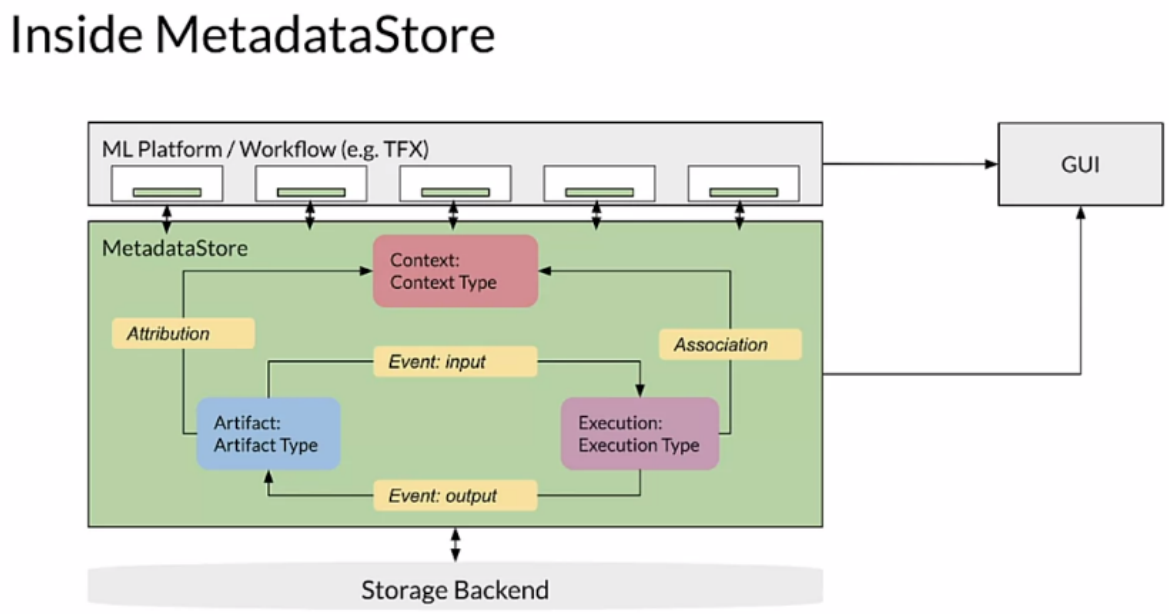
So, ML metadata stores a wide range of information about the results of the components and execution runs of a pipeline. It stores artifacts and it stores the executions of each component in the pipeline. It also stores the lineage information for each artifact that is generated. All of this information is represented in metadata objects and this metadata is stored in a back end storage solution.

Metadata stored



Let's take a look at the architecture of ML metadata or MLMD. On the top are the various components present in any ML pipeline. All of these components are individually connected to a centralized metadata store of ML metadata, so that each component can independently access the metadata at any stage of the pipeline.

An ML pipeline may optionally have a gooey console that can access the data from the metadata store directly to track the progress of each component. At the heart of the metadata store is the **artifact** which is described by its corresponding artifact type. Artifacts become the inputs to any pipeline components which depend on them. And the corresponding use of artifacts by components is recorded in **executions**. The input of an artifact into a component is described by an input event and the corresponding output of a new artifact from the component is described by an **output event**. This interaction between artifacts and executions is represented by **context** through the relationships of **attribution** and **association**. Lastly, all of the data generated by the metadata store is stored in various types of back end storage like SQLite and MySQL and large objects are stored in a file system or block store.



You learned a lot about the architecture and nomenclature of ML metadata or MLMD and the artifacts and entities which it contains. This should give you some idea of how you can leverage MLMD to track metadata and the results flowing through your pipeline to better understand your training process, both now and in previous training runs of your pipeline.

Key points

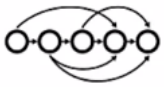
ML metadata:

- Architecture and nomenclature
- Tracking metadata flowing between components in pipeline

ML Metadata in Action

Besides data lineage and provenance tracking, you get several other benefits through ML Metadata. This includes the ability to construct a directed acyclic graph or DAG, of the component executions occurring in a pipeline, which can be useful for debugging purposes. Through this, you can verify which inputs have been used in an execution. You can also summarize all the artifacts belonging to a specific type generated after a series of experiments. For example, you can list all of the models that have been trained. You can then compare them to evaluate your various training runs.

Other benefits of ML Metadata



Produce DAG of pipelines

Verify the inputs used in an execution

List all artifacts

Compare artifacts

You may have to install ML Metadata, which you can do using PIP as shown here. Then there are two imports from ML Metadata which are used frequently, the ML Metadata store itself and the ML Metadata store PB2, which is a protocol buffer or protobuf.

Import ML Metadata

```
!pip install ml-metadata


from ml_metadata import metadata_store
from ml_metadata.proto import metadata_store_pb2
```

Start by setting up the storage backend. ML Metadata store is the database where ML Metadata registers all of the metadata associated with your project. ML Metadata provides APIs to connect with a fake database for quick prototyping, SQLite and MySQL. We also need a block store or file system where ML Metadata stores large objects like datasets.

ML Metadata storage backend

- ML metadata registers metadata in a database called Metadata Store
- APIs to record and retrieve metadata to and from the storage backend:
  - Fake database: in-memory for fast experimentation/prototyping
  - SQLite: in-memory and disk
  - MySQL: server based
  - Block storage: File system, storage area network, or cloud based

Let's quickly explore the first three options. For any storage backend, you'll need to create a connection config object using the metadata store PB2 protobuf. Then, based on your choice of storage backend, you need to configure this connection object. Here you're signaling that your fake database is in parent, which is the primary memory of the system on which it's running.

Finally, you create the store object passing in the connection config. Regardless of what storage or database you use, the store object is a key part of how you interact with ML Metadata.

Fake database

```
connection_config = metadata_store_pb2.ConnectionConfig()

# Set an empty fake database proto
connection_config.fake_database.SetInParent()


store = metadata_store.MetadataStore(connection_config)
```

To use SQLite, you start by creating a connection config again then you configure the connection config object with the location of your SQLite file. Make sure to provide that with the relevant read and write permissions based on your application. Finally, you create the store object passing in the connection config.

SQLite

```
connection_config = metadata_store_pb2.ConnectionConfig()

connection_config.sqlite.filename_uri = '...'
connection_config.sqlite.connection_mode = 3 # READWRITE_OPENCREATE

store = metadata_store.MetadataStore(connection_config)
```

As you might imagine, using MySQL is very similar, you start by creating a connection config. Your connection config object should be configured with the relevant host name, the port number, the database location, username, and password of your MySQL database user and that's shown here. Finally, you create the store object passing in the connection config.

MySQL

```
connection_config = metadata_store_pb2.ConnectionConfig()

connection_config.mysql.host = '...'
connection_config.mysql.port = '...'
connection_config.mysql.database = '...'
connection_config.mysql.user = '...'
connection_config.mysql.password = '...'

store = metadata_store.MetadataStore(connection_config)
```

Let's review the key points which we covered in this lesson. First, you learned about data lineage and provenance to address data evolution over the ML Pipeline lifecycle. Then you went over metadata for tracking those data changes. Then you inspected the architecture of the ML Metadata library. Finally, in the ungraded lab, you read an example to register artifacts, executions, and contexts.

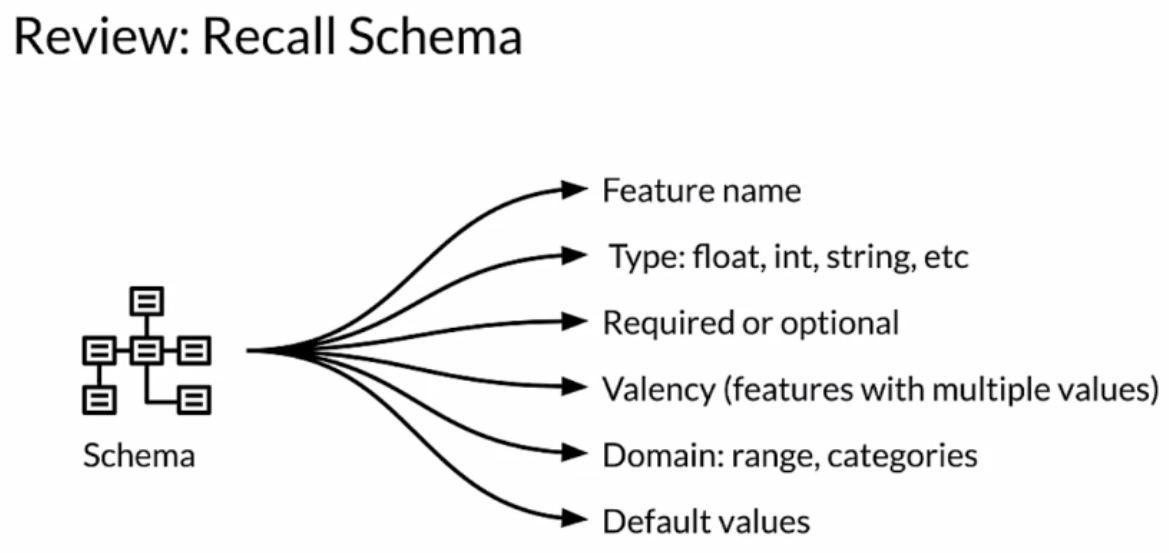
Key points

- Walk through over the data journey addressing lineage and provenance
- The importance of metadata for tracking data evolution
- ML Metadata library and its usefulness to track data changes
- Running an example to register artifacts, executions, and contexts

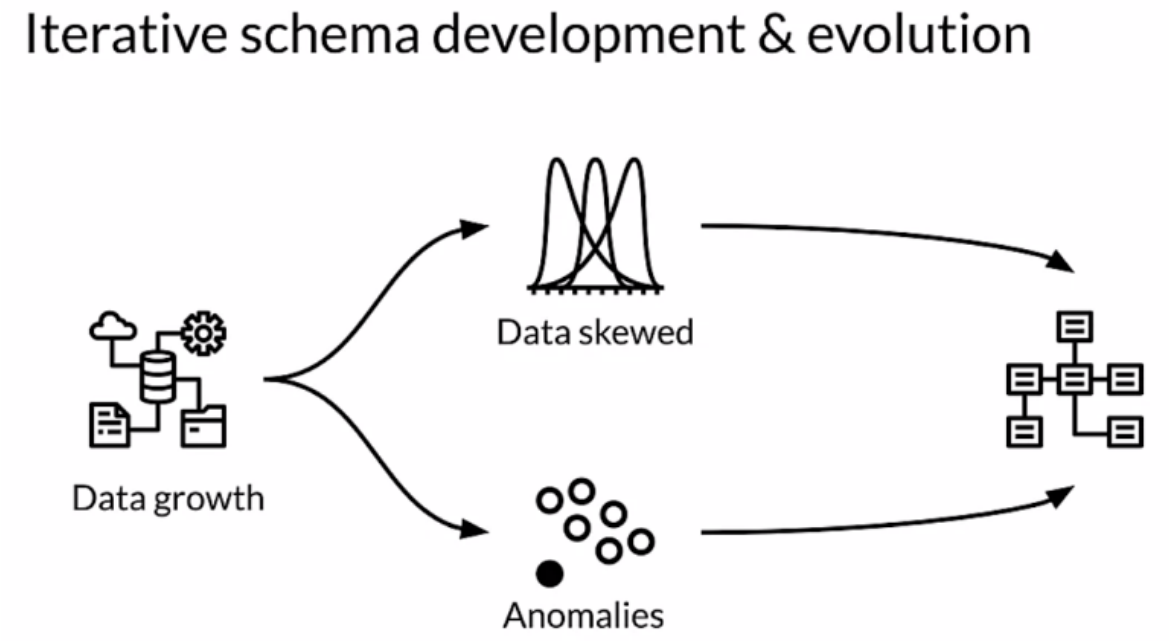
Evolving Data

Schema Development

Schemas are relational objects summarizing the features in a given dataset or project. They include the feature name, the feature of variable type. For example, an integer, float, string or categorical variable. Whether or not the feature is required. The valency of the feature, which applies to features with multiple values like lists or array features, and expresses the minimum and maximum number of values. Information about the range and categories and feature default values.

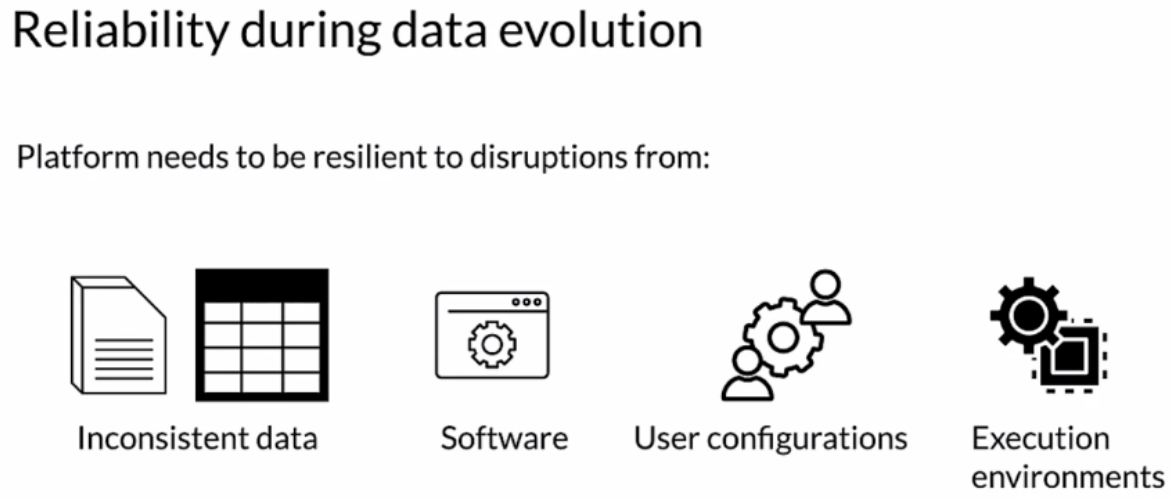


Schemas are important, as your data and feature set evolves over time. From your experience, you know that data keeps changing and this change often results in change distributions. Let's focus on how to observe data that has changed as it keeps evolving. Changing data often results in a new schema being generated. However, there are some special use cases. Imagine that even before you assess the dataset, you have an idea or information about the expected range of values for your features. The initial dataset that you've received is covering only a small range of those values. In that case, it makes sense to adjust or create your schema to reflect the expected range of values for your features. A similar situation may exist for the expected values of categorical features. Besides that, your schema can help you find problems or anomalies in your dataset, such as missing required values or values of the wrong type. All these factors have to be considered when you're designing the schema of your ML pipeline.

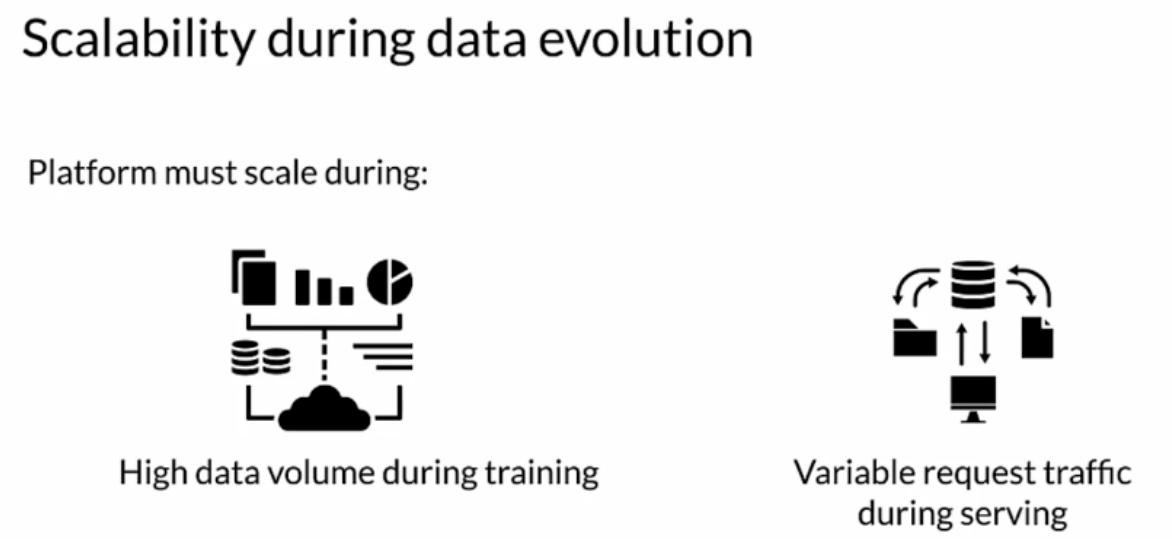


As data keeps evolving, there are some requirements which must be met in production deployments. Let's consider some important ones.

The first factor is **reliability**. The ML platform of your choice should be resilient to disruptions from inconsistent data. There's no guarantee that you'll receive clean and consistent data every time. In fact, I can almost guarantee you that you won't. Your system needs to be designed to handle that efficiently. Also, your software might generate unexpected runtime errors and your pipeline needs to gracefully handle that also. Problems with misconfiguration should be detected and handled gracefully. Above all, the orchestration of execution among different components in the pipeline should happen smoothly and efficiently.

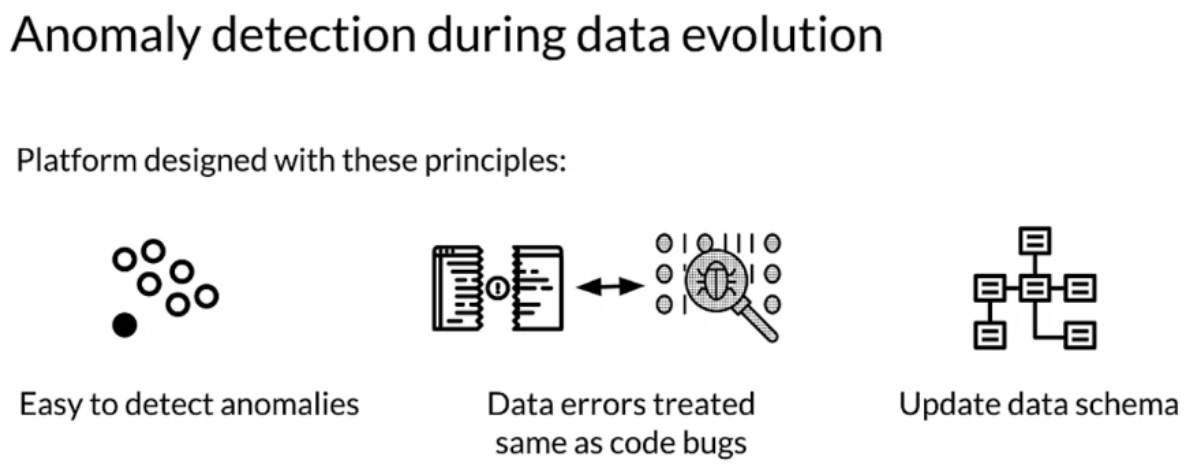


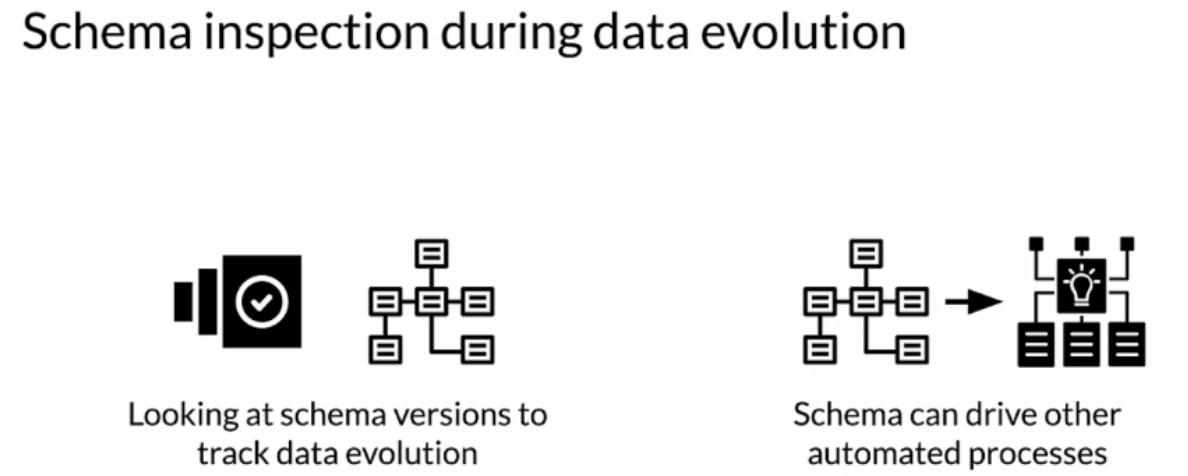
Another factor to consider in your ML pipeline platform is **scalability** during data evolution. During training, your pipeline may need to handle a large amount of training data well, including keeping expensive accelerators like GPUs and TPUs busy. When you serve your model, especially in online scenarios such as running on a server, you'll almost always have varying levels of request traffic. Your infrastructure needs to scale up and down to meet those latency requirements while minimizing the cost.



If your system isn't designed to handle data evolution, it will quickly run into problems. These include the introduction of **anomalies in your dataset**. Will your system detect those anomalies? **Your system and your development process should be designed to treat data errors as**

first-class citizens in the same way that bugs in your code are treated. In some cases, those anomalies are alerting you that you need to update the schema and accommodate valid changes in your data. The evolution of your schemas can be a useful tool to understand and track the evolution of your data. Also, schemas can be used as key inputs into automated processes that work with your data like automatic feature engineering.

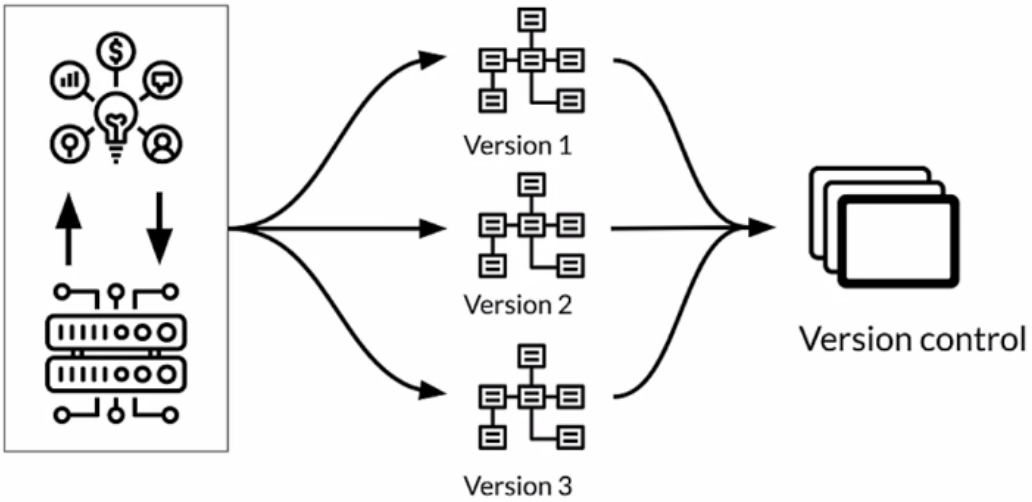




Schema Environments

Your business and data will evolve throughout the lifetime of your production pipeline. It's often the case that as your data evolves, your schema evolves also. As you're developing your code to handle changes in your schema, you may have multiple versions of your schema all active at the same time. You may have one schema being used for development, one currently in test, and another currently in production. Having version control for your schemas, just like you do for your code, helps make this manageable.

Multiple schema versions



In some cases, you may need to have different schemas to support multiple training and deployment scenarios for different data environments. For example, you may want to use the same model on a server and in a mobile application but imagine that a particular feature is different in those two environments. Maybe in one case it's an integer and the other case it's a float. You need to have a different schema for each to reflect the difference in the data. Along with that, your data's evolving. Potentially, in all your different data environments at once. But at the same time you also needed to check your data for problems or anomalies, and schemas are a key part of checking for anomalies.

Maintaining varieties of schema

Let's look at an example of how a schema can help you detect errors in your serving request data and why multiple versions of your schema are important. We'll start by inferring the serving schema and we'll use TensorFlow Data Validation or TFDV to do that. Then we're going to generate statistics for the serving dataset. Then we'll use TFDV to find if there are any problems with this data and visualize the result in a notebook.

Inspect anomalies in serving dataset

```
stats_options = tfdv.StatsOptions(schema=schema,
                                  infer_type_from_schema=True)

eval_stats = tfdv.generate_statistics_from_csv(
    data_location=SERVING_DATASET,
    stats_options=stats_options
)

serving_anomalies = tfdv.validate_statistics(eval_stats, schema)
tfdv.display_anomalies(serving_anomalies)
```

TFDV reports back that there are anomalies in the serving data. Since this is a dataset that contains prediction requests, that's actually not surprising. The label which is cover type is missing, but the schema is telling TFDV that the cover type feature is required. So it's flagging this as an anomaly. How do we fix this problem?

Anomaly: No labels in serving dataset

Feature name	Anomaly short description	Anomaly long description
'Cover_Type'	Out-of-range values	Unexpectedly small value: 0.

In scenarios where you need to maintain multiple types of the same schemas, you often need to keep the schema environment. This is most commonly true of the difference between training and serving data. You can choose to customize your schema based on the situation you're going to handle. For example, in this case, the setup is to maintain two schemas. One for training data, where the label is required and the other for serving where we know we won't have the label.

Schema environments

- Customize the schema for each environment
- Ex: Add or remove label in schema based on type of dataset

The code for multiple schema environments is fairly straightforward. In our existing environment, we already have a schema for training. We then create two named environments called training and serving. We modify our serving environment by removing the cover type feature. Since we know that in serving, we won't have that in our feature set.

Create environments for each schema

```
schema.default_environment.append('TRAINING')
schema.default_environment.append('SERVING')

tfdv.get_feature(schema, 'Cover_Type')
    .not_in_environment.append('SERVING')
```

Lastly, the code sets up the serving environment and uses it to validate the serving data. Now, there are no anomalies found since we're using the correct schema for our data.

Inspect anomalies in serving dataset

```
serving_anomalies = tfdv.validate_statistics(eval_stats,
                                             schema,
                                             environment='SERVING')


tfdv.display_anomalies(serving_anomalies)
# No anomalies found
```

Let's review. First we discussed how to iteratively update and fine tune your schema to adapt to evolving data. Then we focused on reliability and scalability during the evolution cycle of data. Then you implemented schema environments to deal with anomalies in your serving data.

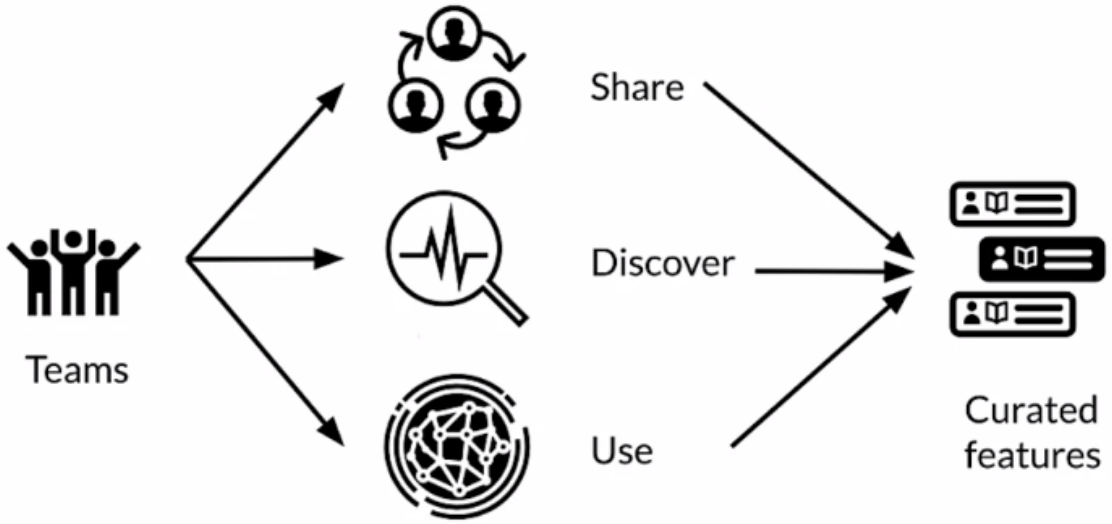
Key points

- Iteratively update and fine-tune schema to adapt to evolving data
- How to deal with scalability and anomalies

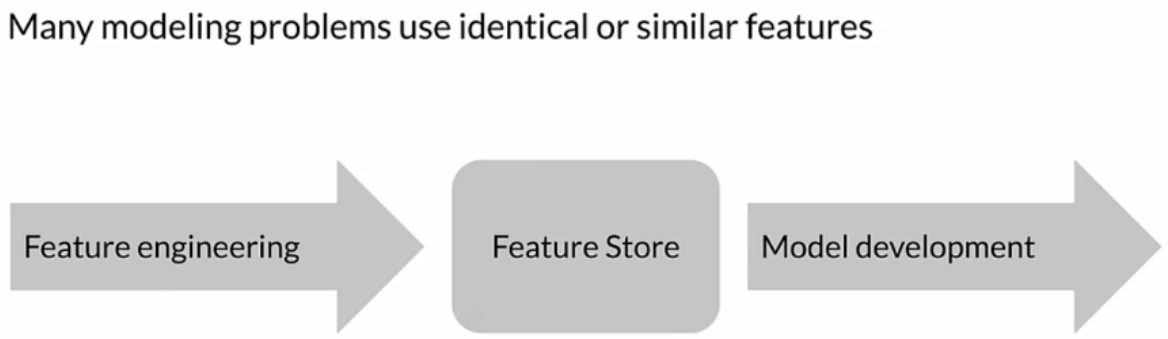
Enterprise Data Storage

Feature Stores

Now let's turn to the question of where we should store our data. We'll start with a discussion of feature stores. A feature store is a central repository for storing documented, curated and access controlled features. Using a feature store enables teams to share, discover and use highly curated features. A feature store makes it easy to discover and consume that feature and that can be both online or offline for both serving and training.

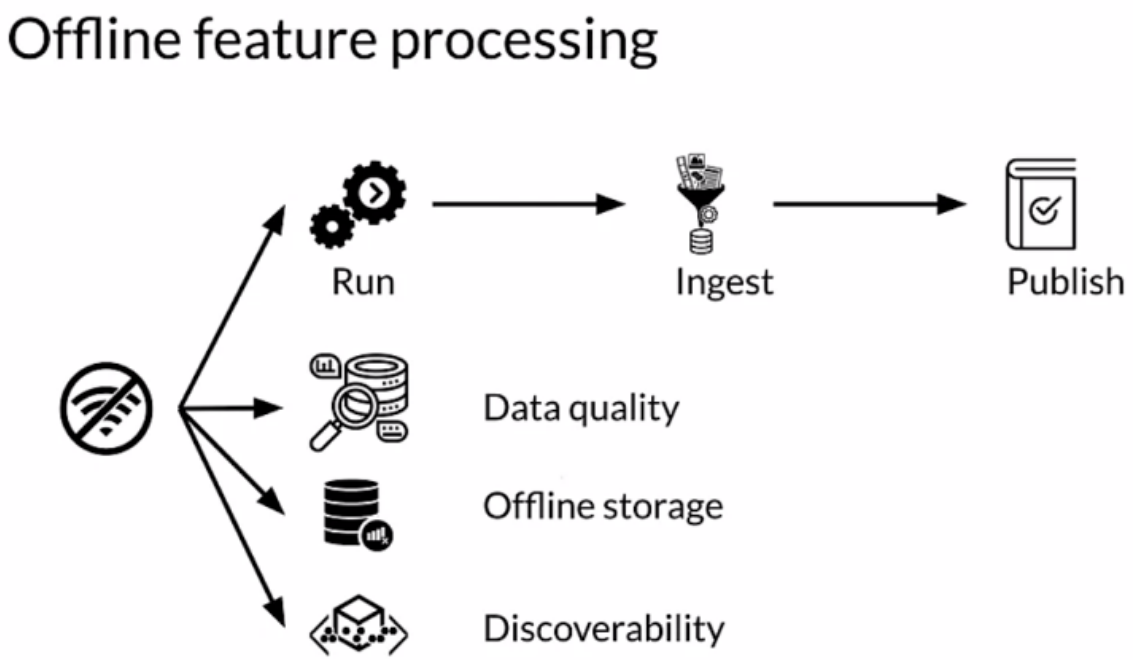


But people often discover that many modeling problems use identical or similar features. So often the same data is used in multiple modeling scenarios. In many cases, a feature store can be seen as the interface between feature engineering and model development. Feature stores are valuable centralized feature repositories that reduce redundant work. They are also valuable because they enable teams to share data and discover data that is already available. You may have different teams in an organization with different business problems that they're trying to solve. But they're using identical data or data that's very similar. For these reasons, feature stores are becoming the predominant choice for enterprise data storage. For machine learning, for large projects and organizations.

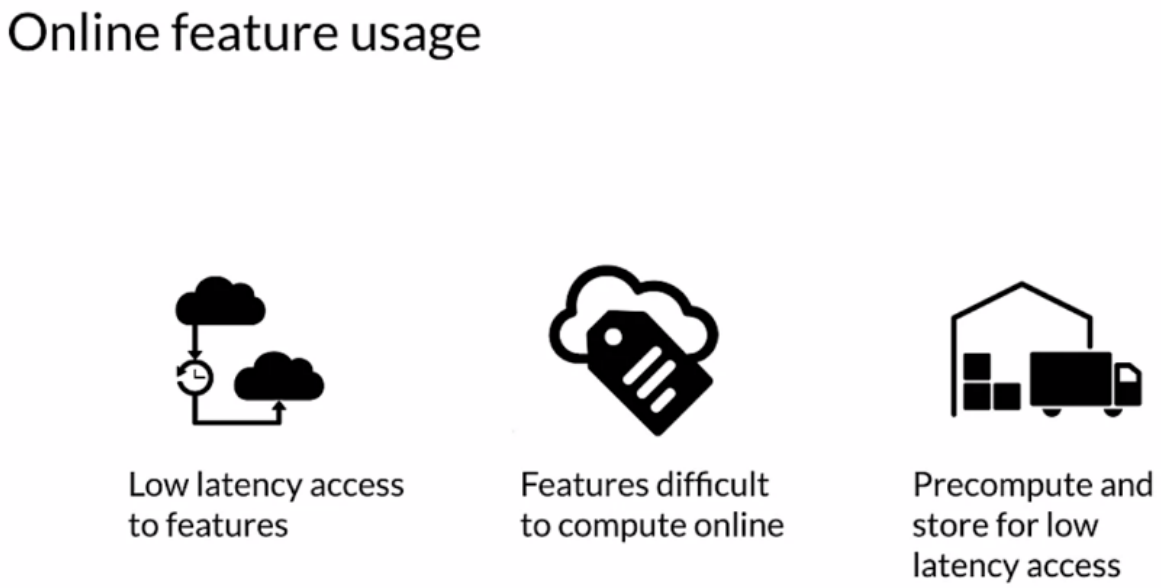


Feature stores often allow transformations of data so that you can avoid duplicating that processing in different individual pipelines. The access to the data in feature stores can be controlled based on role based permissions. The data in the feature stores can be aggregated to form new features. It can also be anonymous sized and even purged for things like wipeouts For GDP our compliance, for example.

Feature stores typically allow for feature processing offline that can be on a regular basis, maybe on a cron job, for example. Imagine that you're going to run a job to ingest data. And then maybe do some feature engineering on it and produce additional features from it. Maybe for feature crosses, for example, these new features will also be published to the Feature store. You can also integrate that with monitoring tools as your processing and adjusting your data. You could be running monitoring again offline. Those processed features are stored for offline use. They can also be part of a prediction request. But doing a join with the data provided in the prediction request to pull in additional information. Feature metadata allows you to discover the features that you need. The metadata that describes the data that you are keeping is a tool. And often the main tool for trying to discover the data that you're looking for.

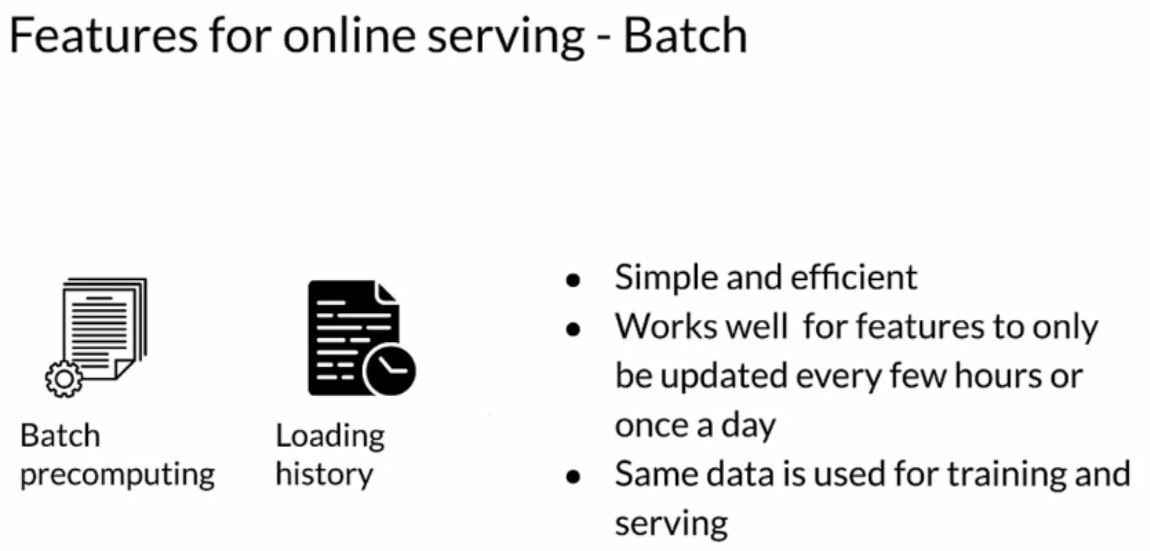


For online feature usage where predictions must be returned in real time. The latency requirements are typically fairly strict. You're going to need to make sure that you have fast access to that data. If you're going to do a join, for example, maybe with user account information along with individual requests. That join has to happen quickly. That's good, but it's often difficult to compute some of those features in a performant manner online. So having pre computed features is often a good idea. If you pre compute and store those features then you can use them later. And typically that's at fairly low latency.

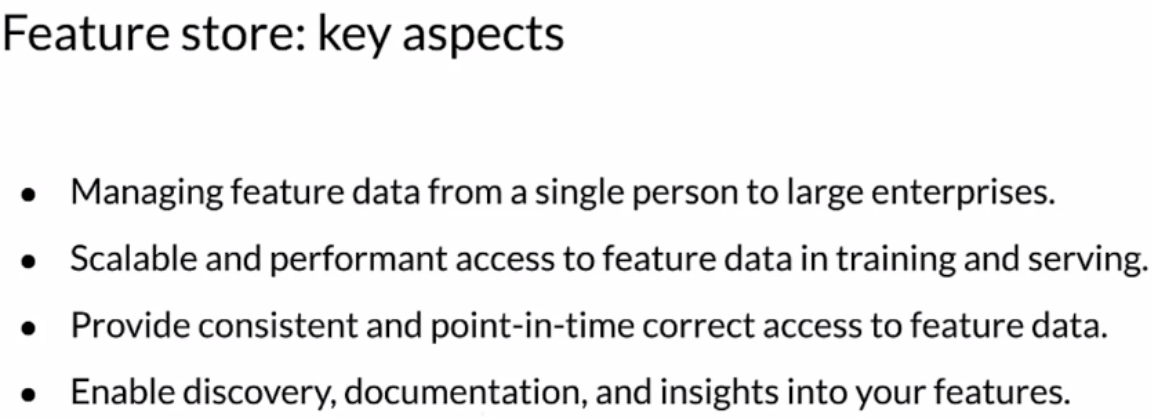


You can also do this in a batch environment. Again, you don't want the latency to be too long, but it probably isn't as strict as an online request. For pre-computing and loading, especially things like historical features, it tends to be fairly simple. For historical features in a badger environment,

it's also usually straightforward. However, when you're training your model, you need to make sure that you only include data that will be available at the time that a serving request is made. Including data that is only available at some time after a serving request is referred to as time travel. And many feature stores include safeguards to avoid that. Besides it violates the laws of physics and we don't want to do that. You might do pre computing on a clock every few hours or once a day. You're going to use of course, that same data for both training and serving in order to avoid training, serving skew.

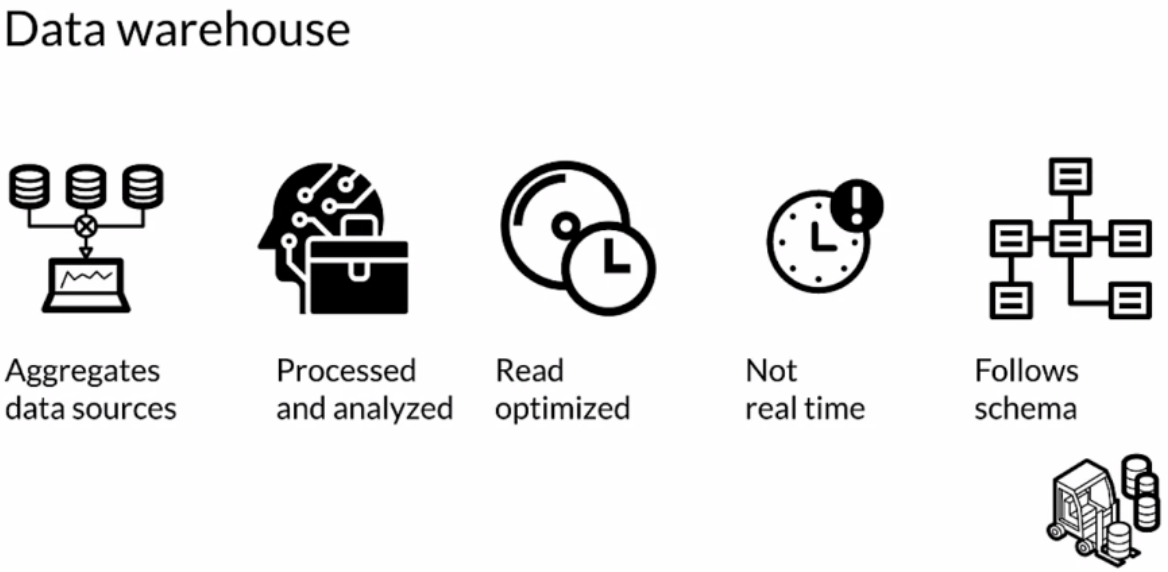


The goals of most feature stores are providing a unified means of managing featured data. They can scale from a single person up to large enterprises. It needs to be performant and you want to try to use that same data, both when you're training and serving your models. You want consistency and also point in time correct access to feature data. You want to avoid making a prediction, for example, using data that will only be available in the future when you're serving your model. In other words, if you're trying to predict something that will happen tomorrow. You want to make sure that you are not including data from tomorrow. It should only be data from before tomorrow. Most feature stores provide tools to enable discovery and to allow you to document and provide insights into your features.

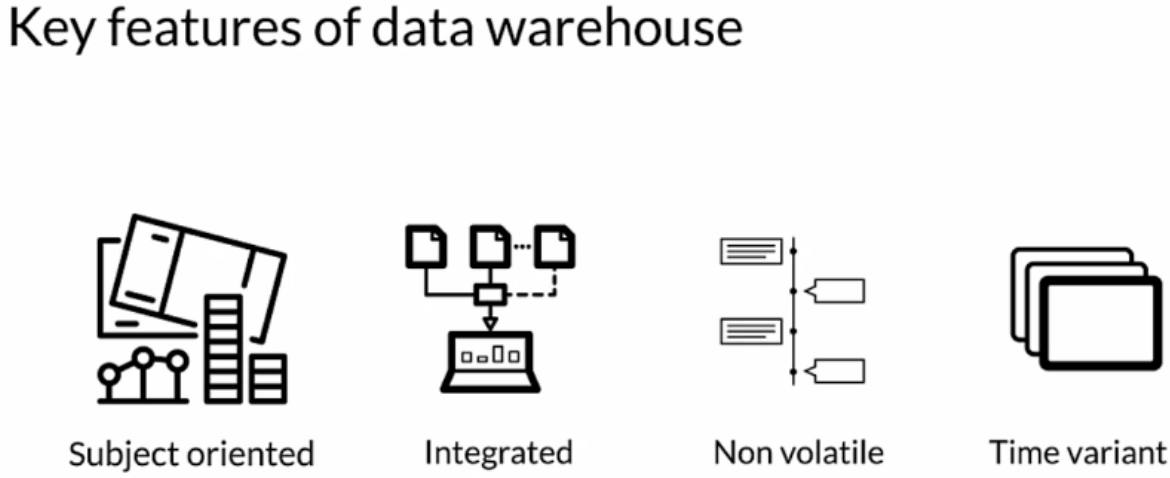


Data Warehouse

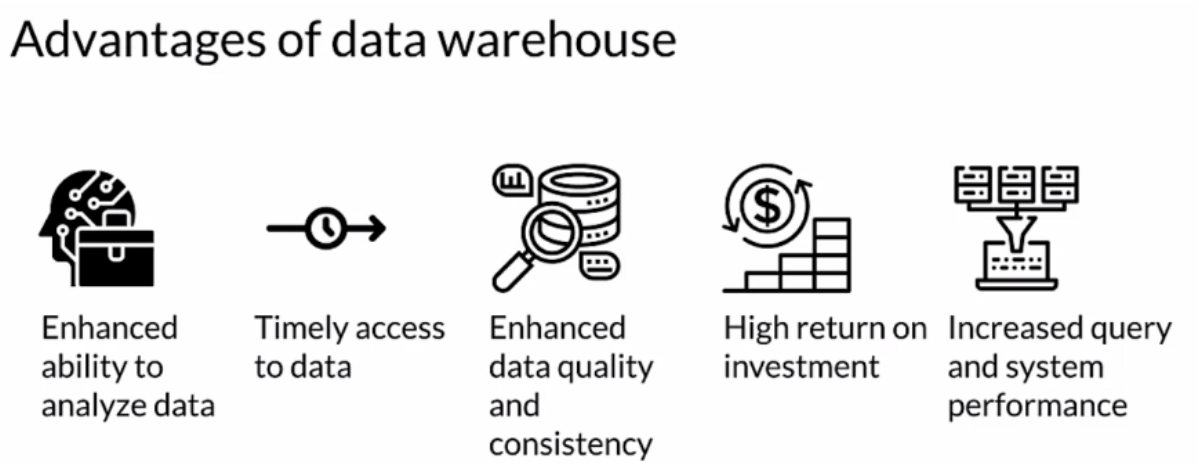
Data warehouses were originally developed for big data and business intelligence applications, but they're also valuable tools for production ML. A data warehouse is a technology that aggregates data from one or more sources so that it can be processed and analyzed. A data warehouse is usually meant for long running batch jobs and their storage is optimized for read operations. Data entering into the warehouse may not be in real time. When you're storing data in a data warehouse, your data needs to follow a consistent schema.



Let's look at some key features of data warehouses. A data warehouse is subject oriented and the information that's stored in it revolves around a topic. For example, data stored in a data warehouse may be focused on the organization's customers or vendors, or etc. The data in data warehouse may be collected from multiple types of sources, such as relational databases or files and so forth. The data collected in a data warehouse is usually timestamped to maintain the context of when it was generated. Data warehouses are nonvolatile, which means the previous versions of data are not erased when new data is added. That means that you can access the data stored in a data warehouse as a function of time and understand how that data has evolved.



Let's look at some advantages of data warehouses. First, data warehouses offer enhanced ability to analyze your data by time stamping your data. A data warehouse can help maintain contexts. When you store your data in a data warehouse, it follows a consistent schema and that helps improve the data quality and consistency. Studies have shown that the return on investment for data warehouses tend to be fairly high for many use cases. Lastly, the read and query efficiency from data warehouses is typically high, giving you fast access to your data.



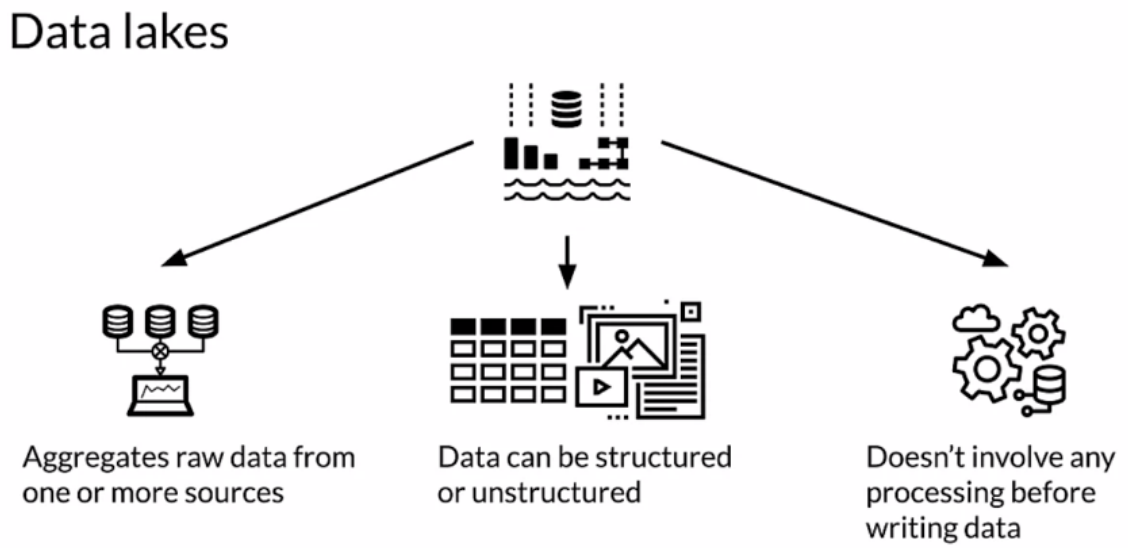
A natural question is, what's the difference between a data warehouse and a database? Here are some comparisons. Data warehouses are meant for analyzing data, whereas databases are often used for transaction purposes. Inside a data warehouse, there may be a delay between storing the data and the data getting reflected in the system. But in a database, data is usually available immediately after it's stored. Data warehouses store data as a function of time, and therefore, historical data is also available. Data warehouses are typically capable of storing a larger amount of data compared to databases. Queries in data warehouses are complex in nature and tend to run for a long time. Whereas queries in databases are simple and tend to run in real time. Normalization is not necessary for data warehouses, but it should be used with databases.

Comparison with databases

Data warehouse	Database
Online analytical processing (OLAP)	Online transactional processing (OLTP)
Data is refreshed from source systems	Data is available real-time
Stores historical and current data	Stores only current data
Data size can scale to >= terabytes	Data size can scale to gigabytes
Queries are complex, used for analysis	Queries are simple, used for transactions
Queries are long running jobs	Queries executed almost in real-time
Tables need not be normalized	Tables normalized for efficiency

Data Lakes

Now, let's discuss another popular enterprise storage solution, data lakes. A data lake is a system or repository of data stored in its natural and raw format, which is usually in the form of blobs or files. A data lake, like a data warehouse, aggregates data from various sources of enterprise data. A data lake can include structured data like racial databases or semi-structured data like CSV files, or unstructured data like a collection of images or documents, and so forth. Since data lake store data in its raw format, they don't do any processing, and they usually don't follow a schema.



How does a data lake differ from a data warehouse? Let's compare the two. The primary difference between them is that in a data warehouse, data is stored in a consistent format which follows a schema, whereas in data lakes, the data is usually in its raw format. In data lakes, the reason for storing the data is often not determined ahead of time. This is usually not the case for a data warehouse, where it's usually stored for a particular purpose. Data warehouses are often used by business professionals as well, whereas data lakes are typically used only by data professionals such as data scientists. Since the data in data warehouses is stored in a consistent format, changes to the data can be complex and costly. Data lakes however are more flexible, and make it easier to make changes to the data.

	Data warehouses	Data lakes
Data Structure	Processed	Raw
Purpose of data	Currently in use	Not yet determined
Users	Business professionals	Data scientists
Accessibility	More complicated and costly to make changes	Highly accessible and quick to update

Let's review our discussion of enterprise data storage. First, you learned about feature stores, which are repositories for highly curated feature data, specifically for use in machine learning. You learned about subject-oriented read-optimized data repositories, known as data warehouses. Finally, you explored data lakes, which are repositories for raw and unprocessed data. We've come to the end of another week and we talked about a lot of great stuff this week. ML metadata, working with storage for features stores, schemas, data lakes, and data warehouses. I hope you had fun. I sure did. We'll see you next time.

Key points

- **Feature store**: central repository for storing documented, curated, and access-controlled features, specifically for ML.
- **Data warehouse**: subject-oriented repository of structured data optimized for fast read.
- **Data lakes**: repository of data stored in its natural and raw format.

Week 3 Optional References

Week 3: Data Journey and Data Storage

If you wish to dive more deeply into the topics covered this week, feel free to check out these optional references. You won't have to read these to complete this week's practice quizzes.

Data Versioning:

1. https://dvc.org/
2. https://git-lfs.github.com/

ML Metadata:

1. https://www.tensorflow.org/tfx/guide/mlmd#data_model
2. https://www.tensorflow.org/tfx/guide/understanding_custom_components

Chicago taxi trips data set:

1. https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew/data
2. https://archive.ics.uci.edu/ml/datasets/covertype

Feast:

1. https://cloud.google.com/blog/products/ai-machine-learning/introducing-feast-an-open-source-feature-store-for-machine-learning
2. https://github.com/feast-dev/feast
3. https://blog.gojekengineering.com/feast-bridging-ml-models-and-data-efd06b7d1644

Week 4: Advanced Labeling, Augmentation and Data Preprocessing

Advanced Labeling

Data Augmentation

Preprocessing Different Data Types

Course Resources